Quiz 4

Question 1: Multiple Answer

Average Score 1.47368 points

Which of the are throughput (or bandwidth) metrics?

Correct	Answers	Percent Correct	Percent Incorrect
	Cycle time	94.737%	5.263%
	Instruction Count	96.491%	3.509%
☑	Instructions/second	94.737%	5.263%
	(Instructions/sec)/Watt	91.228%	8.772%
☑	Frames per second.	87.719%	12.281%

What is true about the following x86 instructions:

1. pushl %eax
2. ret
3. enter
4. movl %eax, %ebx
5. movl %eax, -4(%ebx,%ecx,4)

Correct	Answers	Percent Correct	Percent Incorrect
☑	Only one of them can be translated into a single instructions of MIPS assembly.	75.439%	24.561%
	None of them would typically appear in highly-optimized code.	92.982%	7.018%
☑	All of the except #4 access memory.	70.175%	29.825%
☑	Only #1, #3, and #5 modify memory.	71.93%	28.07%
	Only #1, #2, and #5 modify memory	89.474%	10.526%
	All of them access memory.	82.456%	17.544%
☑	#1, #2, #3, #5 all involve arithmetic operations.	77.193%	22.807%

Consider my daily commute: My home is 1.4 miles from the freeway entrance, and there are 6.7 miles of freeway between the entrance and Genesee St, where I exit. My office is 1.6 miles from the freeway. The speed limit on non-freeways is 25MPH. On the freeway, it's 65MPH. I decide that I want to reduce my commute time by going twice as fast as is allowed by law, but I only want to speed on either the freeway or the non-freeway roads. According to Amdahl's law, which one should I choose if I want to save the most time?

Correct		Percent Answered
	Speeding on the freeways.	33.333%
☑	Speeding on the non-freeways.	66.667%
	Speeding on either of them will give the same commute time.	0%
	I want to get this question wrong, so I marked this answer.	0%
	Unanswered	0%

Question 4: Multiple Choice

Average Score 1.40351 points

For a given set of resources (e.g., a network link, or a set of cashiers at the Sunshine store) and some work waiting to be done (e.g., information to be sent over the link, or customers to process), it is usually the case that

Correct		Percent Answered
☑	If utilization is high, then latency is probably long.	70.175%
	If throughput is high, then latency is probably short.	21.053%
	There is no predictable relationship between latency and throughput or utilization.	8.772%
	I love the Sunshine store, and I want to express that love more than I want to get this problem correct.	0%
	Unanswered	0%

Question 5: Multiple Choice

Average Score 2 points

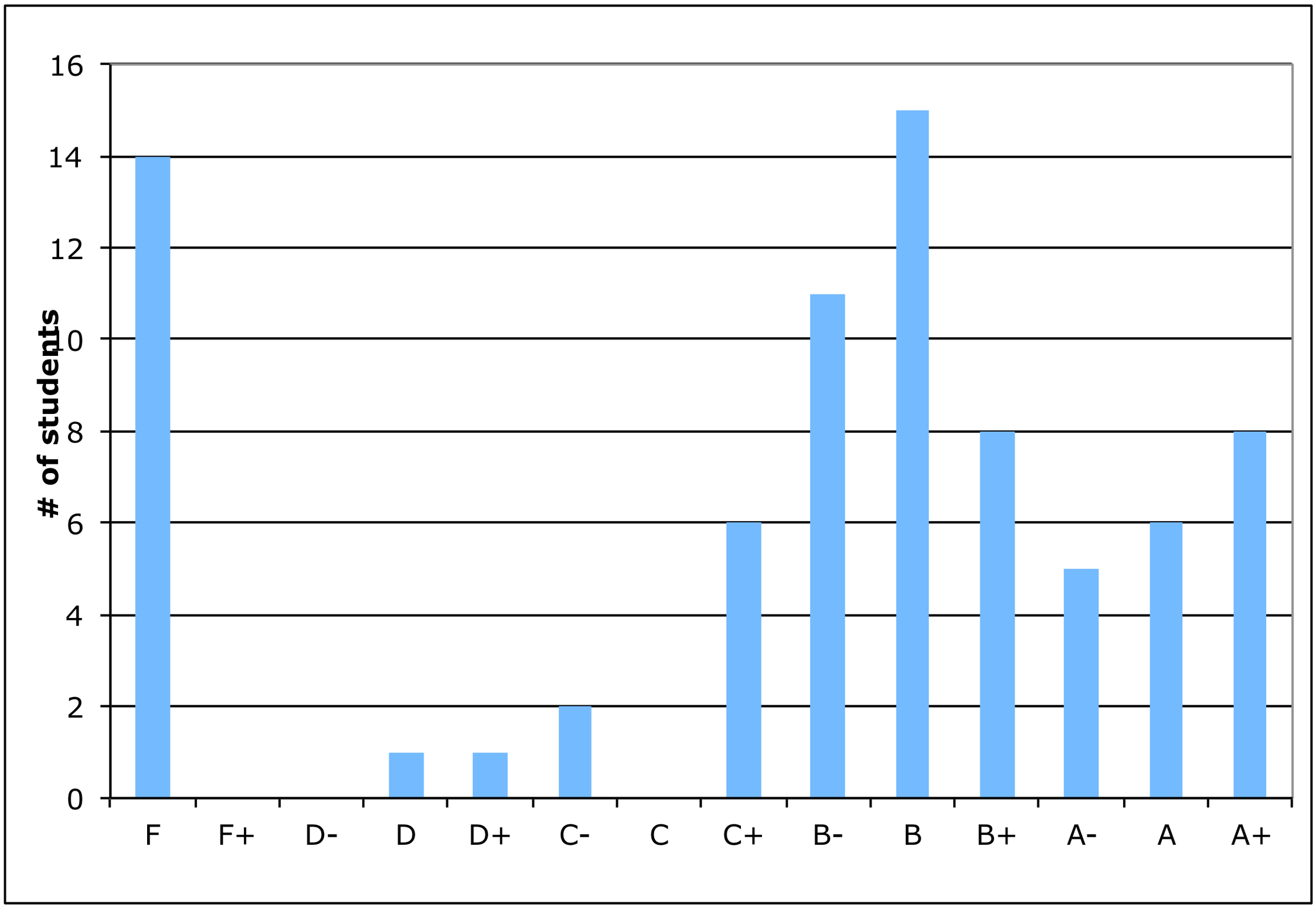
Which instruction does x86 us to load a value from memory into a register.

Correct		Percent Answered
	load	0%
	get	0%
☑	mov	100%
	peek	0%
	Unanswered	0%

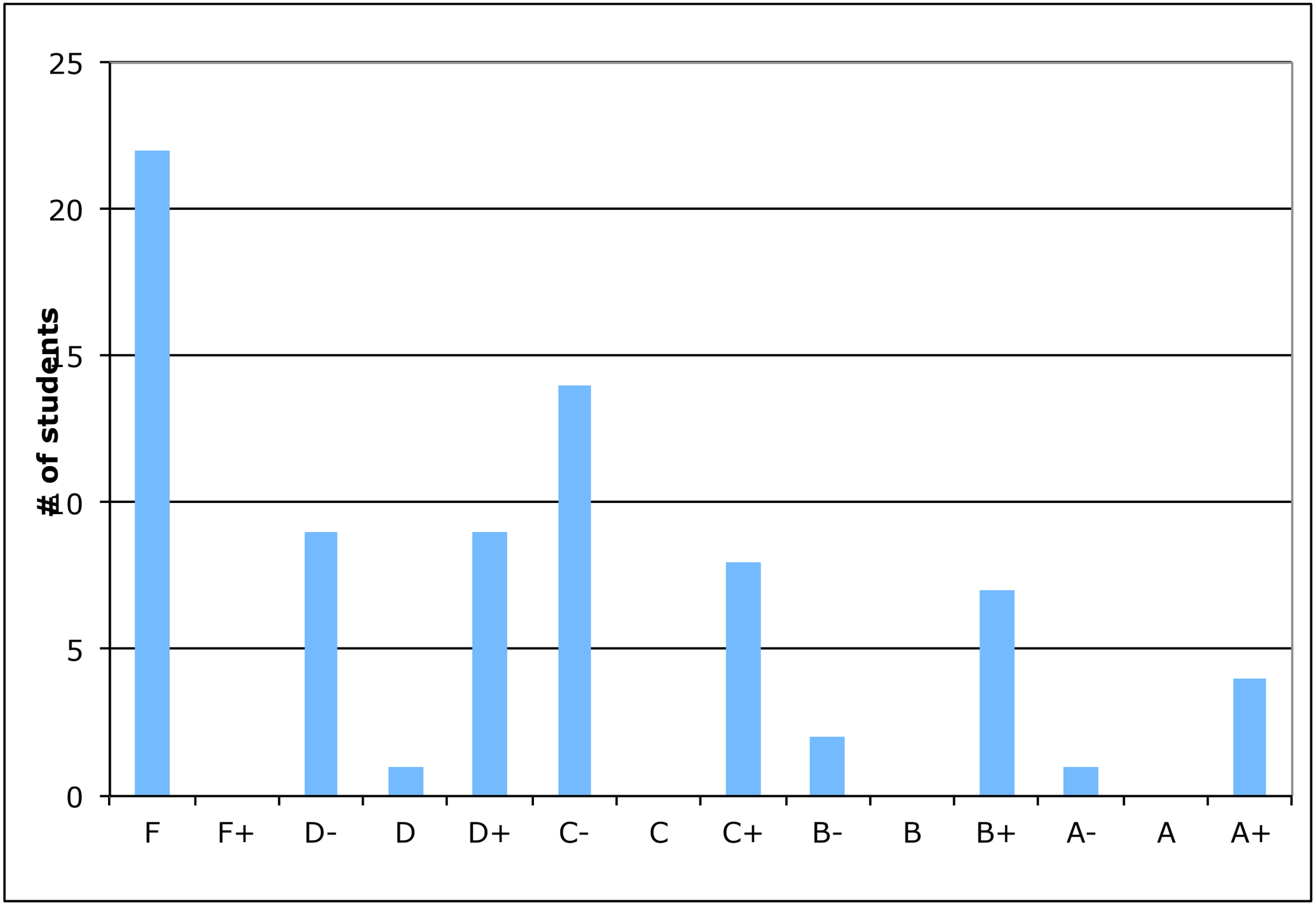
The Midterm is Coming

- Midterm on May 7th.
- Midterm review on May 2nd.
  - Come to class with questions.
- Midterm will cover everything before it
- It will mostly resemble the homeworks and quizzes
- It will be challenging.
- It will be curved.

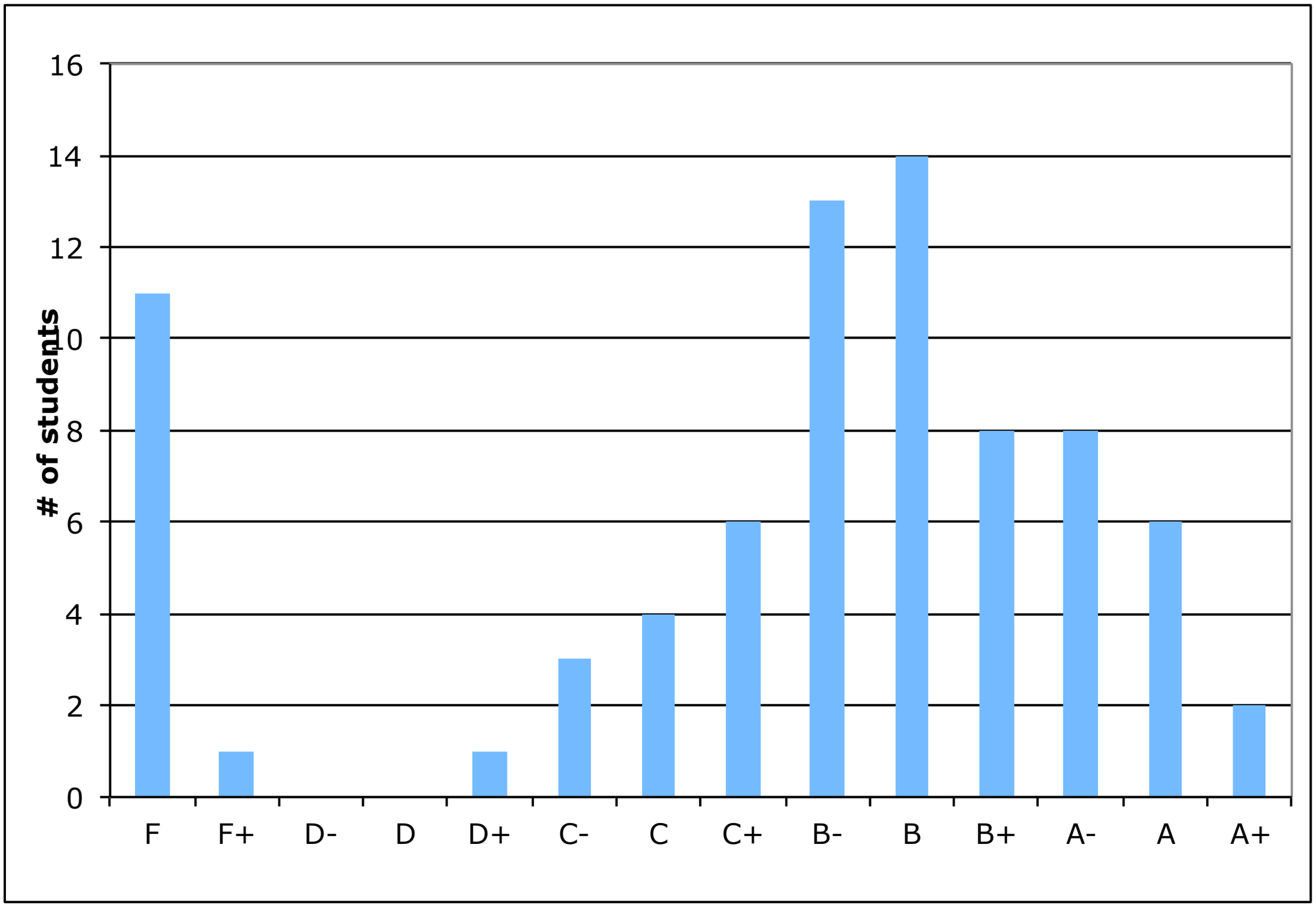
Quiz 1



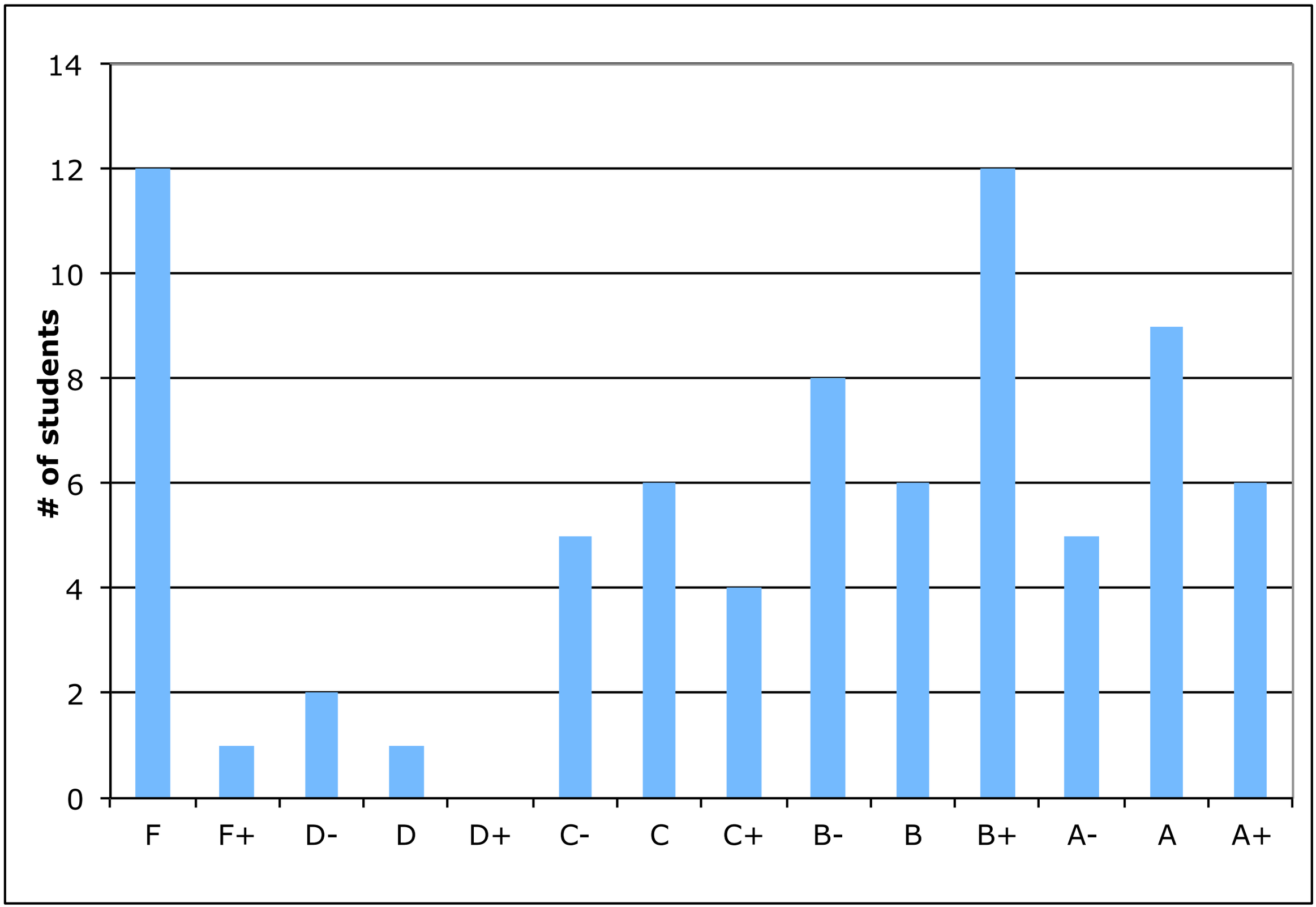
Quiz 2



Quiz 3



HW 2



Implementing a MIPS Processor

Readings: 4.1-4.11

Goals for this Class

- Understand how CPUs run programs
  - How do we express the computation the CPU?
  - How does the CPU execute it?
  - How does the CPU support other system components (e.g., the OS)?
  - What techniques and technologies are involved and how do they work?
- Understand why CPU performance (and other metrics) varies
  - How does CPU design impact performance?
  - What trade-offs are involved in designing a CPU?
  - How can we meaningfully measure and compare computer systems?
- Understand why program performance varies
  - How do program characteristics affect performance?
  - How can we improve a programs performance by considering the CPU running it?
  - How do other system components impact program performance?

Goals

- Understand how the 5-stage MIPS pipeline works
  - See examples of how architecture impacts ISA design
  - Understand how the pipeline affects performance
- Understand hazards and how to avoid them
  - Structural hazards
  - Data hazards
  - Control hazards

Processor Design in Two Acts

Act I: A single-cycle CPU

Foreshadowing

- Act I: A Single-cycle Processor
  - Simplest design – Not how many real machines work (maybe some deeply embedded processors)
  - Figure out the basic parts; what it takes to execute instructions
- Act II: A Pipelined Processor
  - This is how many real machines work
  - Exploit parallelism by executing multiple instructions at once.

Target ISA

- We will focus on part of MIPS
  - Enough to run into the interesting issues
  - Memory operations
  - A few arithmetic/Logical operations (Generalizing is straightforward)
  - BEQ and J
- This corresponds pretty directly to what you'll be implementing in 141L.

Basic Steps for Execution

- Fetch an instruction from the instruction store
- Decode it
  - What does this instruction do?
- Gather inputs
  - From the register file
  - From memory
- Perform the operation
- Write back the outputs
  - To register file or memory
- Determine the next instruction to execute

The Processor Design Algorithm

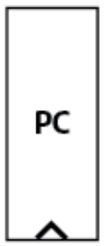
- Once you have an ISA…
- Design/Draw the datapath
  - Identify and instantiate the hardware for your architectural state
  - Foreach instruction
    - Simulate the instruction
    - Add and connect the datapath elements it requires
    - Is it workable? If not, fix it.
- Design the control
  - Foreach instruction
    - Simulate the instruction
    - What control lines do you need?
    - How will you compute their value?
    - Modify control accordingly
    - Is it workable? If not, fix it.
- You've already done much of this in 141L.

- Arithmetic; R-Type
  - Inst = Mem[PC]
  - REG[rd] = REG[rs] op REG[rt]
  - PC = PC + 4

bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6

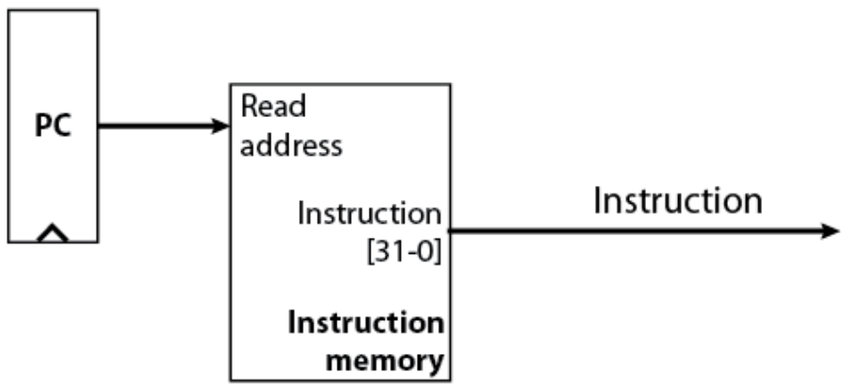
- Arithmetic; R-Type
  - Inst = Mem[PC]
  - REG[rd] = REG[rs] op REG[rt]
  - PC = PC + 4

bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6



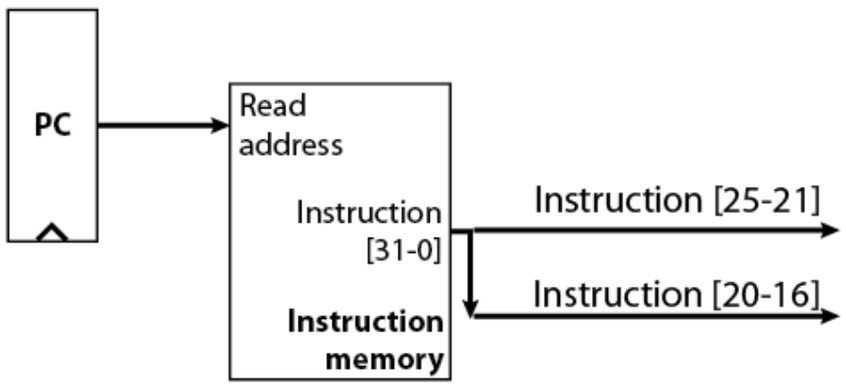
- Arithmetic; R-Type
  - Inst = Mem[PC]
  - REG[rd] = REG[rs] op REG[rt]
  - PC = PC + 4

bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6



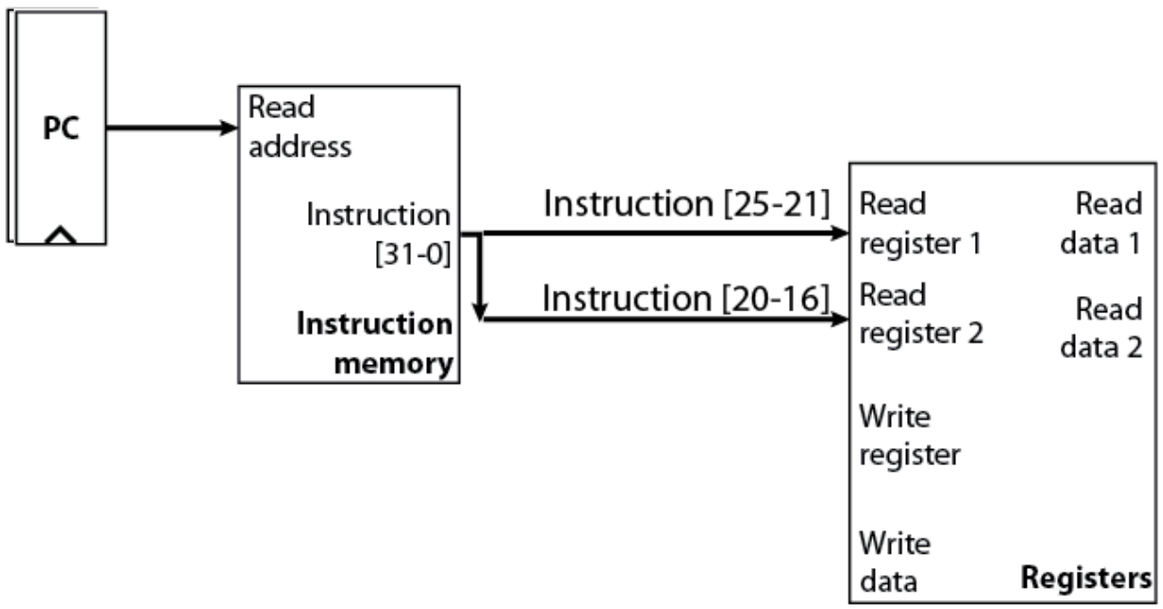
- Arithmetic; R-Type
  - Inst = Mem[PC]
  - REG[rd] = REG[rs] op REG[rt]
  - PC = PC + 4

bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6



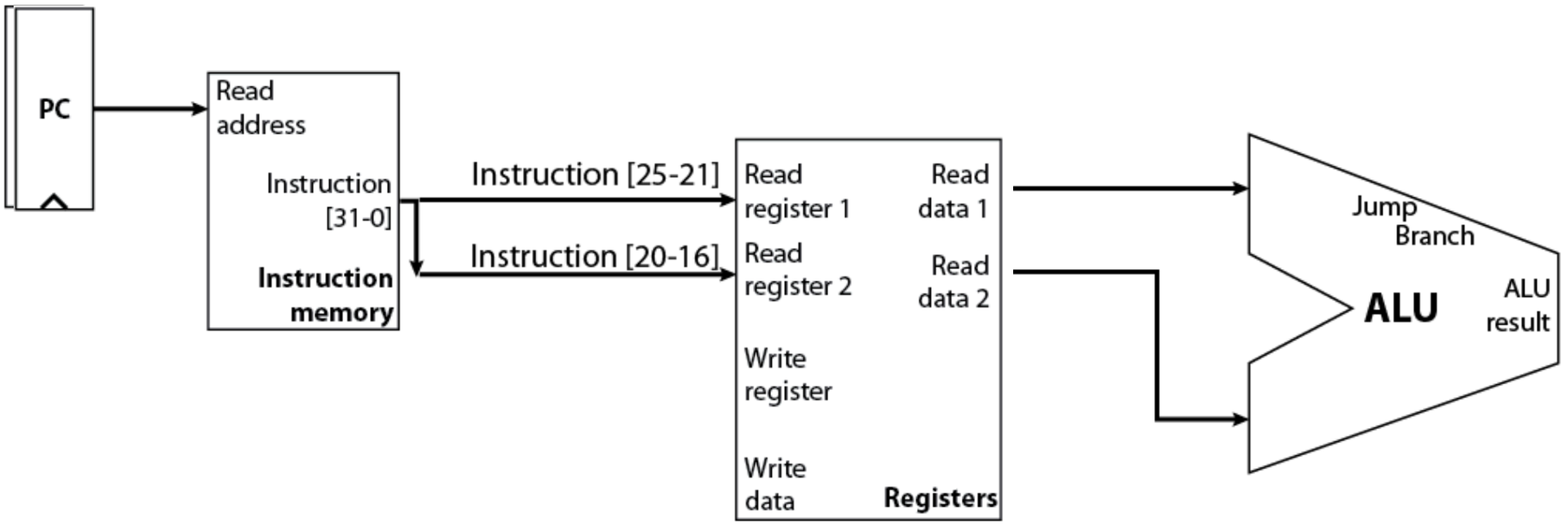
- Arithmetic; R-Type
  - Inst = Mem[PC]
  - REG[rd] = REG[rs] op REG[rt]
  - PC = PC + 4

bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6



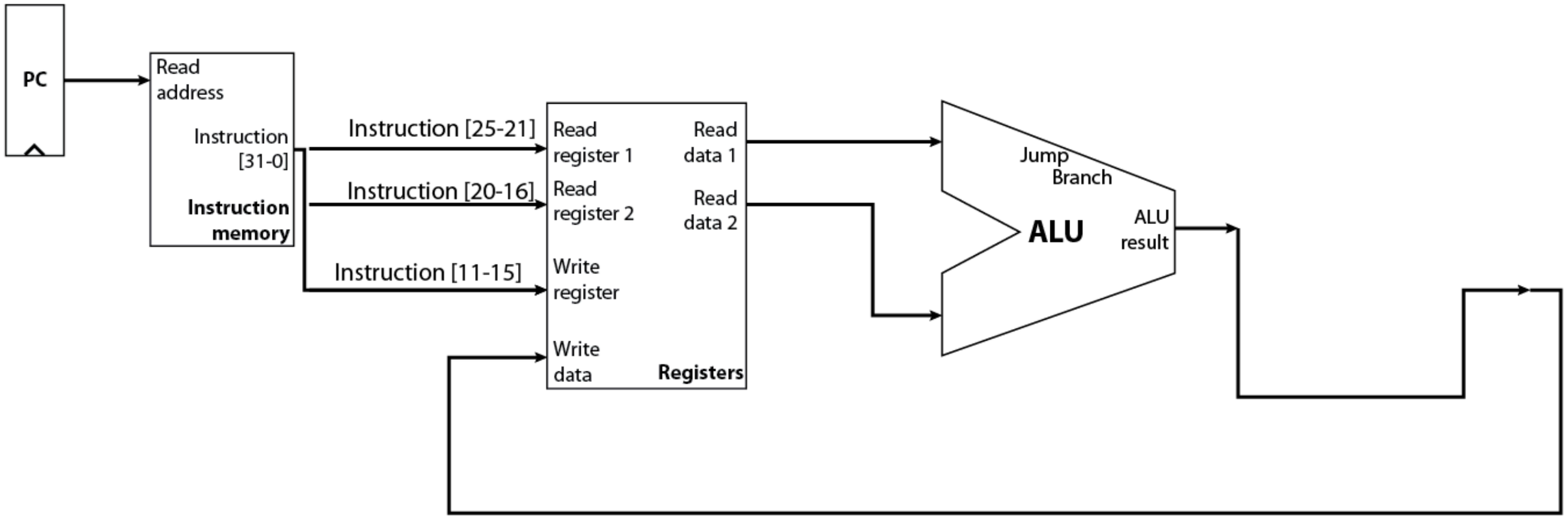
- Arithmetic; R-Type
  - Inst = Mem[PC]
  - REG[rd] = REG[rs] op REG[rt]
  - PC = PC + 4

bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6



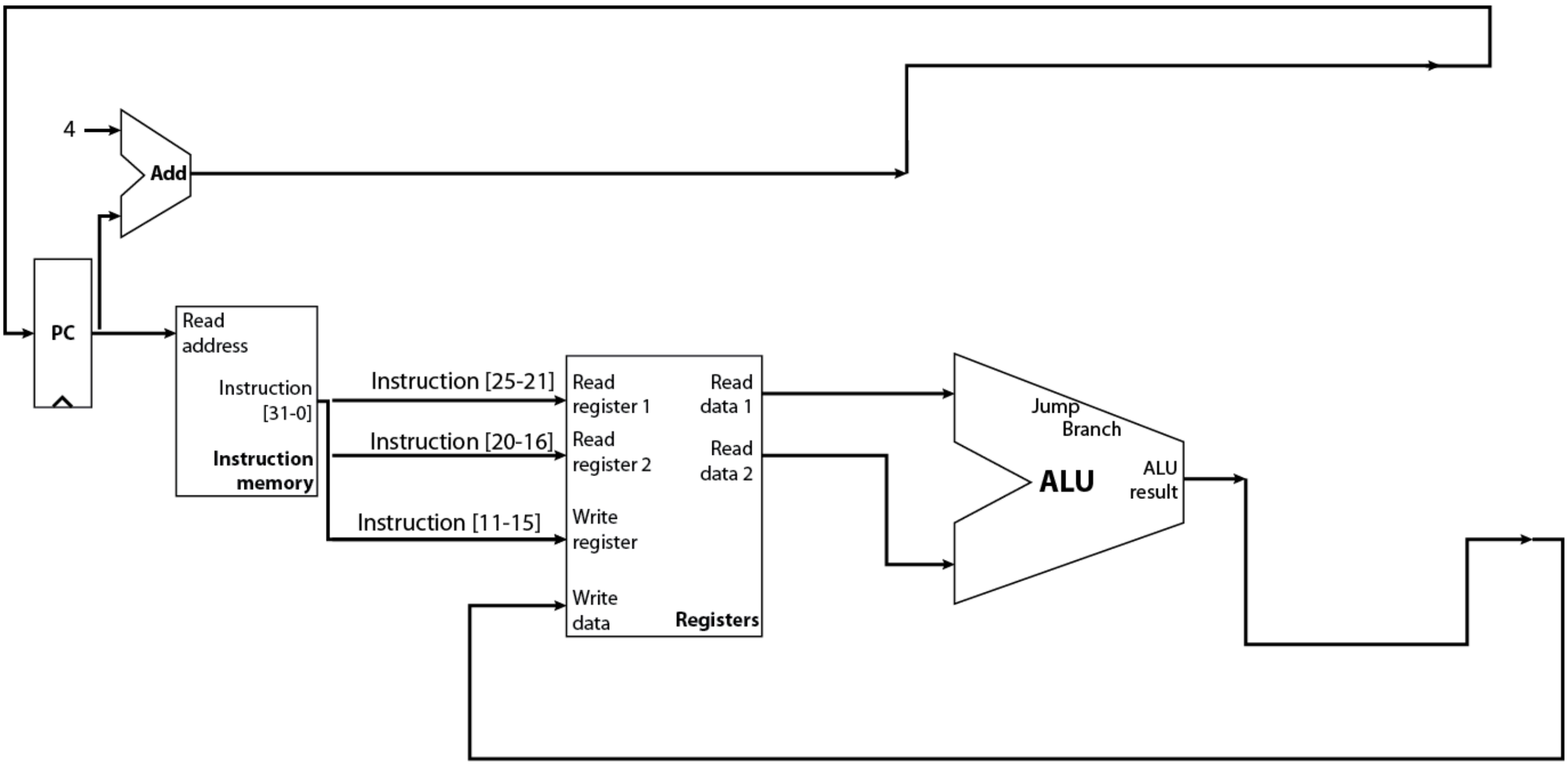
- Arithmetic; R-Type
  - Inst = Mem[PC]
  - REG[rd] = REG[rs] op REG[rt]
  - PC = PC + 4

bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6



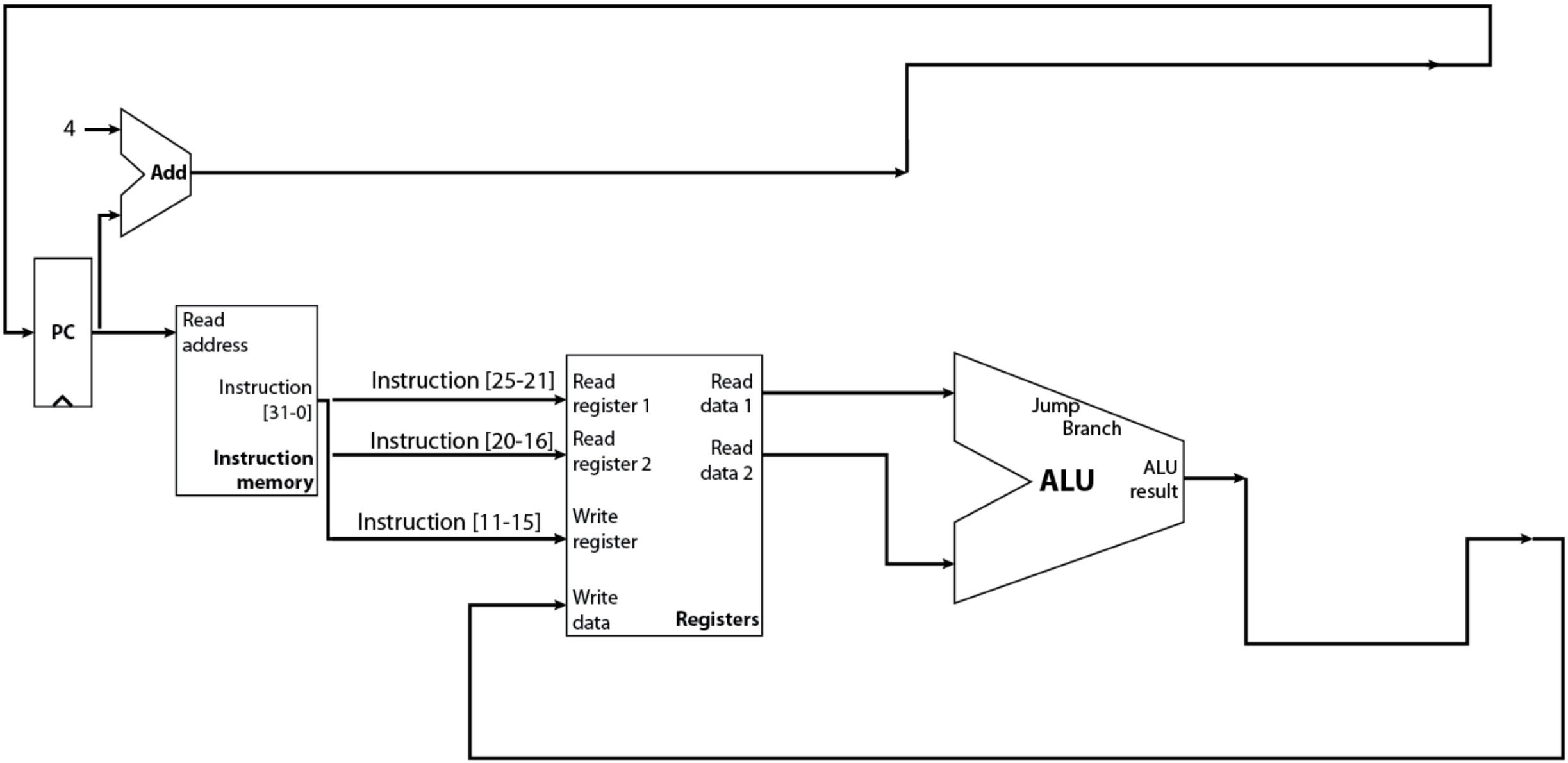
- Arithmetic; R-Type
  - Inst = Mem[PC]
  - REG[rd] = REG[rs] op REG[rt]
  - PC = PC + 4

bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6



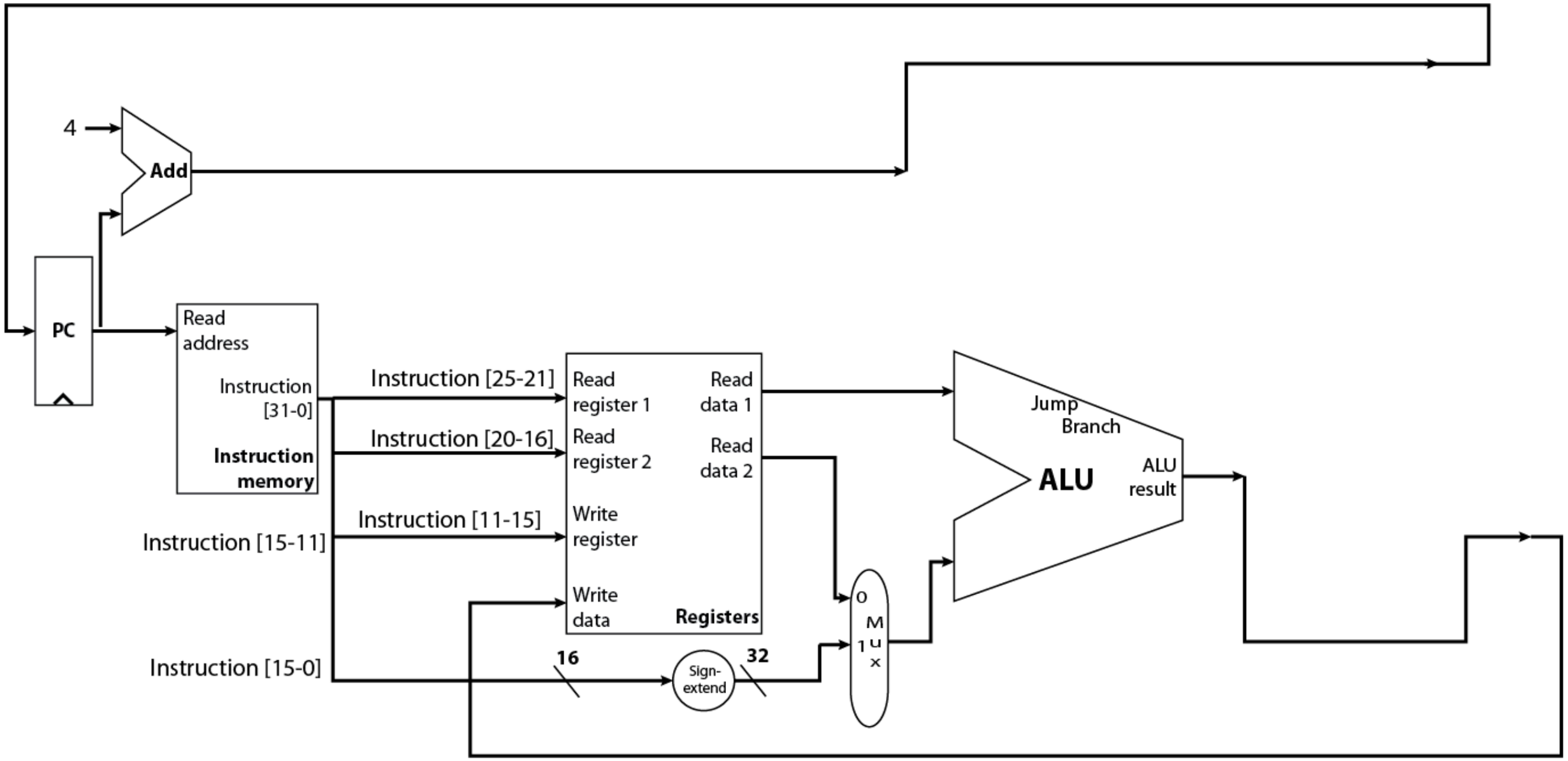
- ADDI; I-Type
  - PC = PC + 4
  - REG[rt] = REG[rs] op SignExtImm

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	imm
# bits	6	5	5	16



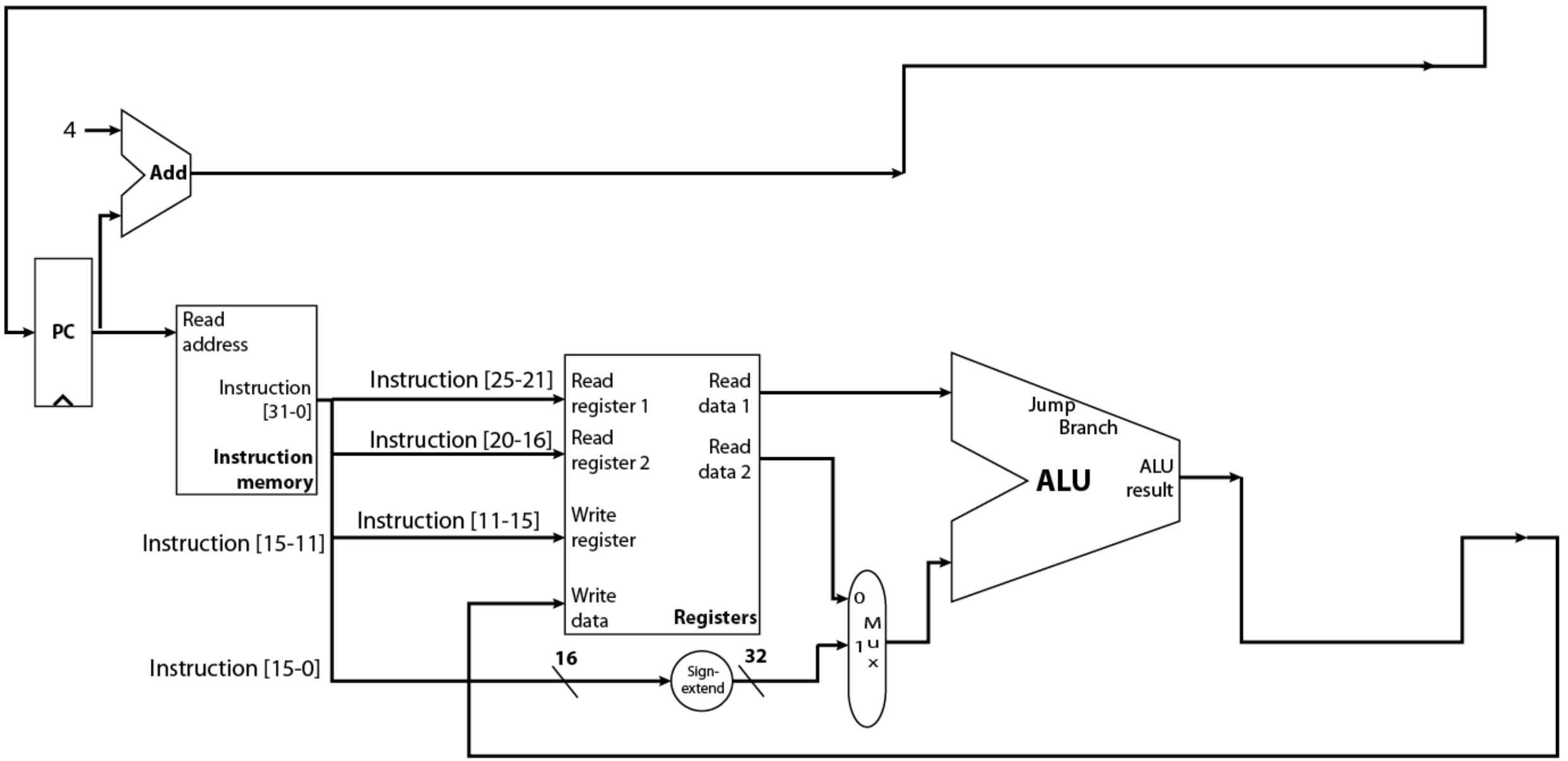
- ADDI; I-Type
  - PC = PC + 4
  - REG[rt] = REG[rs] op SignExtImm

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	imm
# bits	6	5	5	16



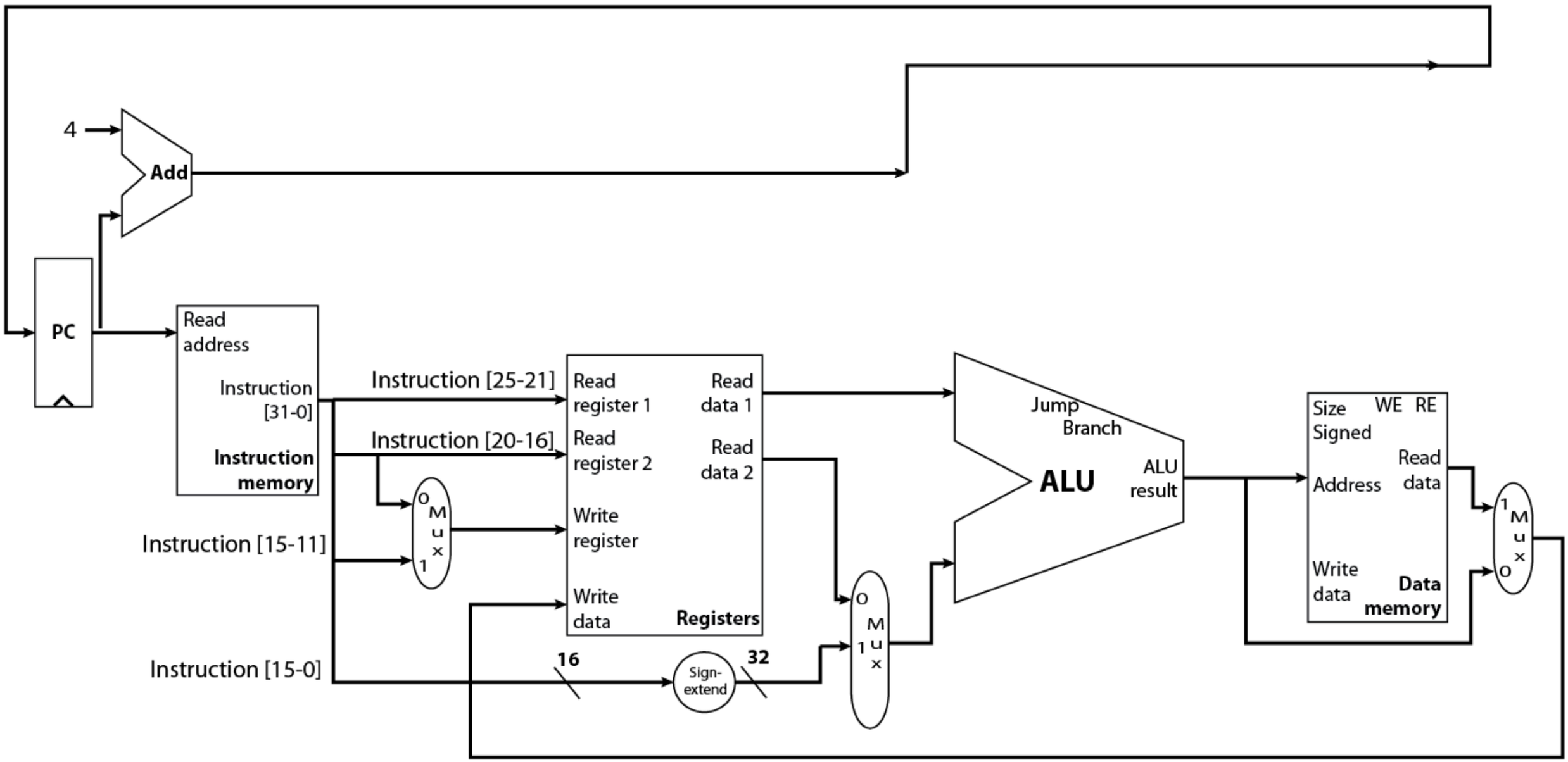
- Load Word
  - PC = PC + 4
  - REG[rt] = MEM[signextendImm + REG[rs]]

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	immediate
# bits	6	5	5	16



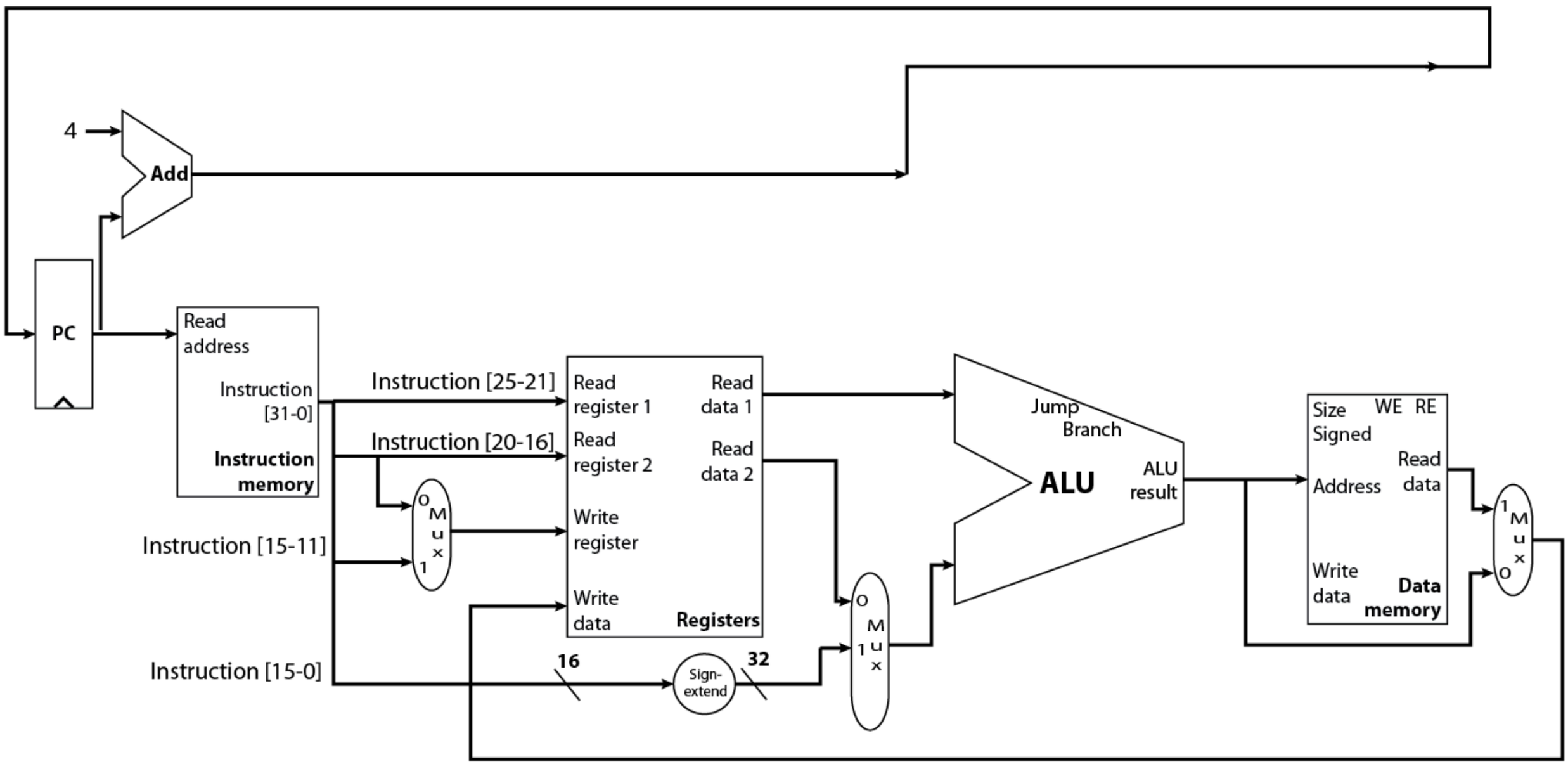
- Load Word
  - PC = PC + 4
  - REG[rt] = MEM[signextendImm + REG[rs]]

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	immediate
# bits	6	5	5	16



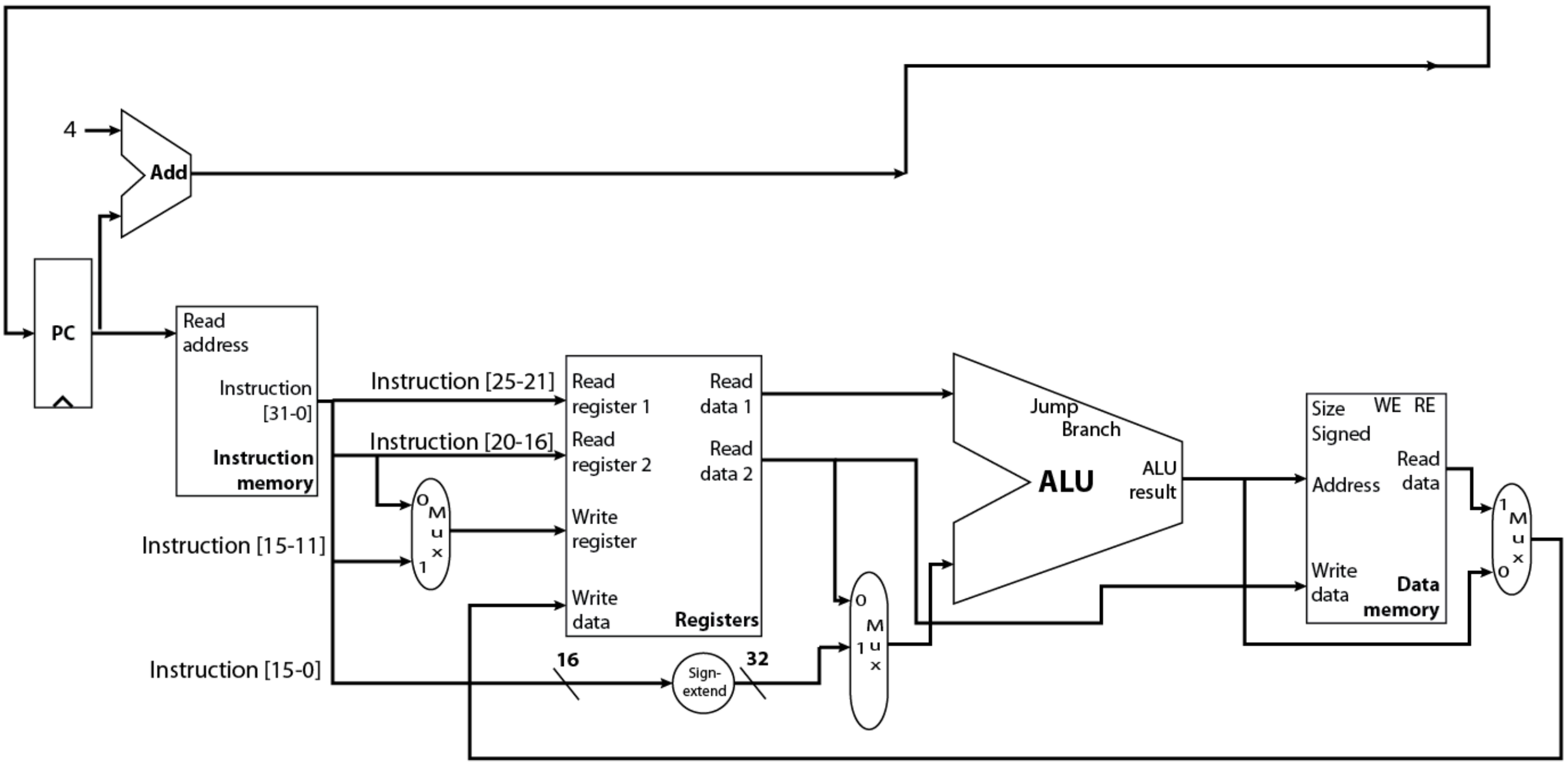
- Store Word
  - PC = PC + 4
  - MEM[signextendImm + REG[rs]] = REG[rt]

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	immediate
# bits	6	5	5	16



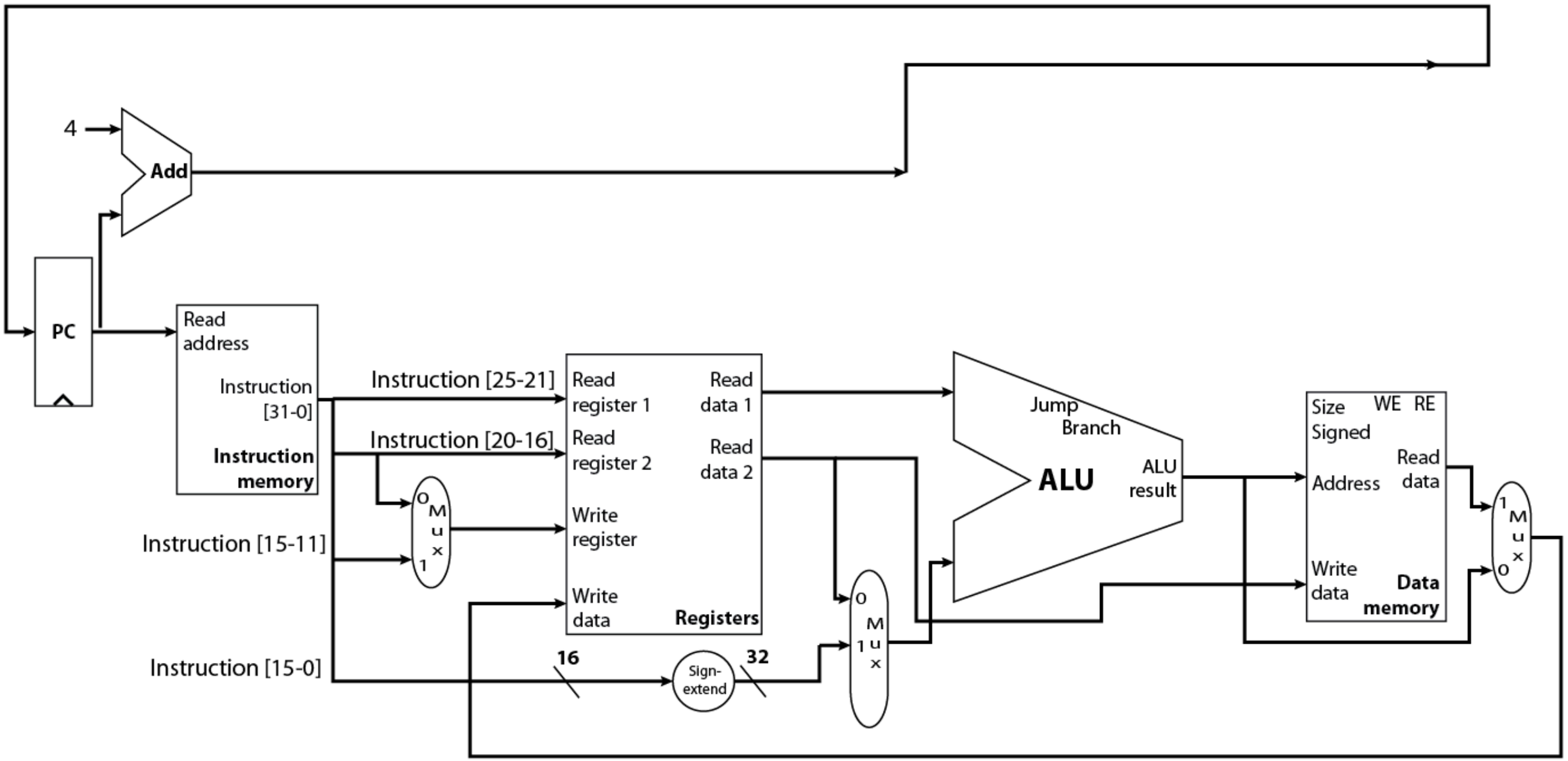
- Store Word
  - PC = PC + 4
  - MEM[signextendImm + REG[rs]] = REG[rt]

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	immediate
# bits	6	5	5	16



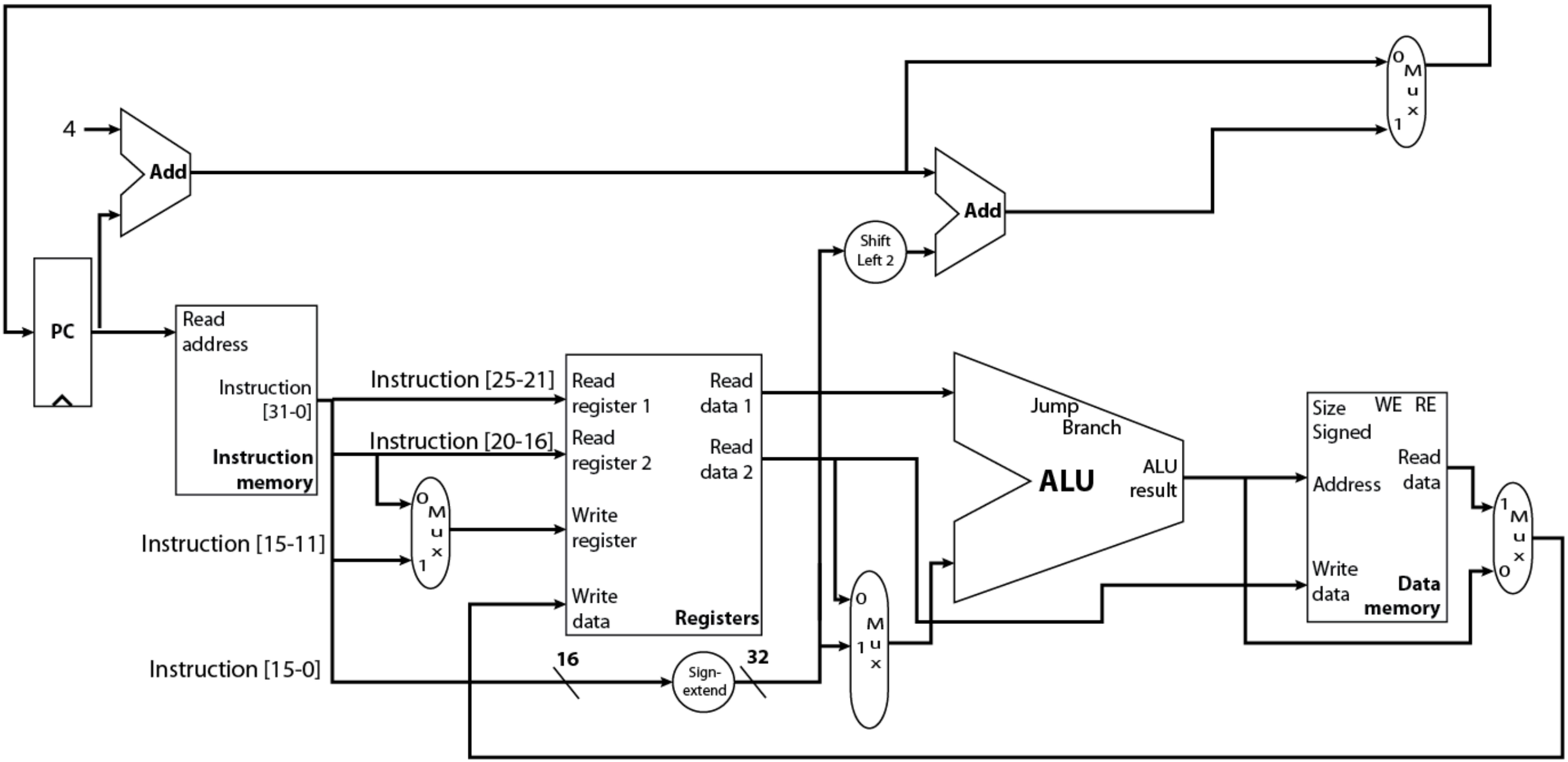
- Branch-equal; I-Type
  - PC = (REG[rs] == REG[rt]) ? PC + 4 + SignExtImmediate *4 : PC + 4;

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	displacement
# bits	6	5	5	16



- Branch-equal; I-Type
  - PC = (REG[rs] == REG[rt]) ? PC + 4 + SignExtImmediate *4 : PC + 4;

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	displacement
# bits	6	5	5	16



A Single-cycle Processor

- Performance refresher
- ET = IC * CPI * CT
- Single cycle ⇒ CPI == 1; That sounds great
- Unfortunately, Single cycle ⇒ CT is large
  - Even RISC instructions take quite a bite of effort to execute
  - This is a lot to do in one cycle

Our Hardware is Mostly Idle

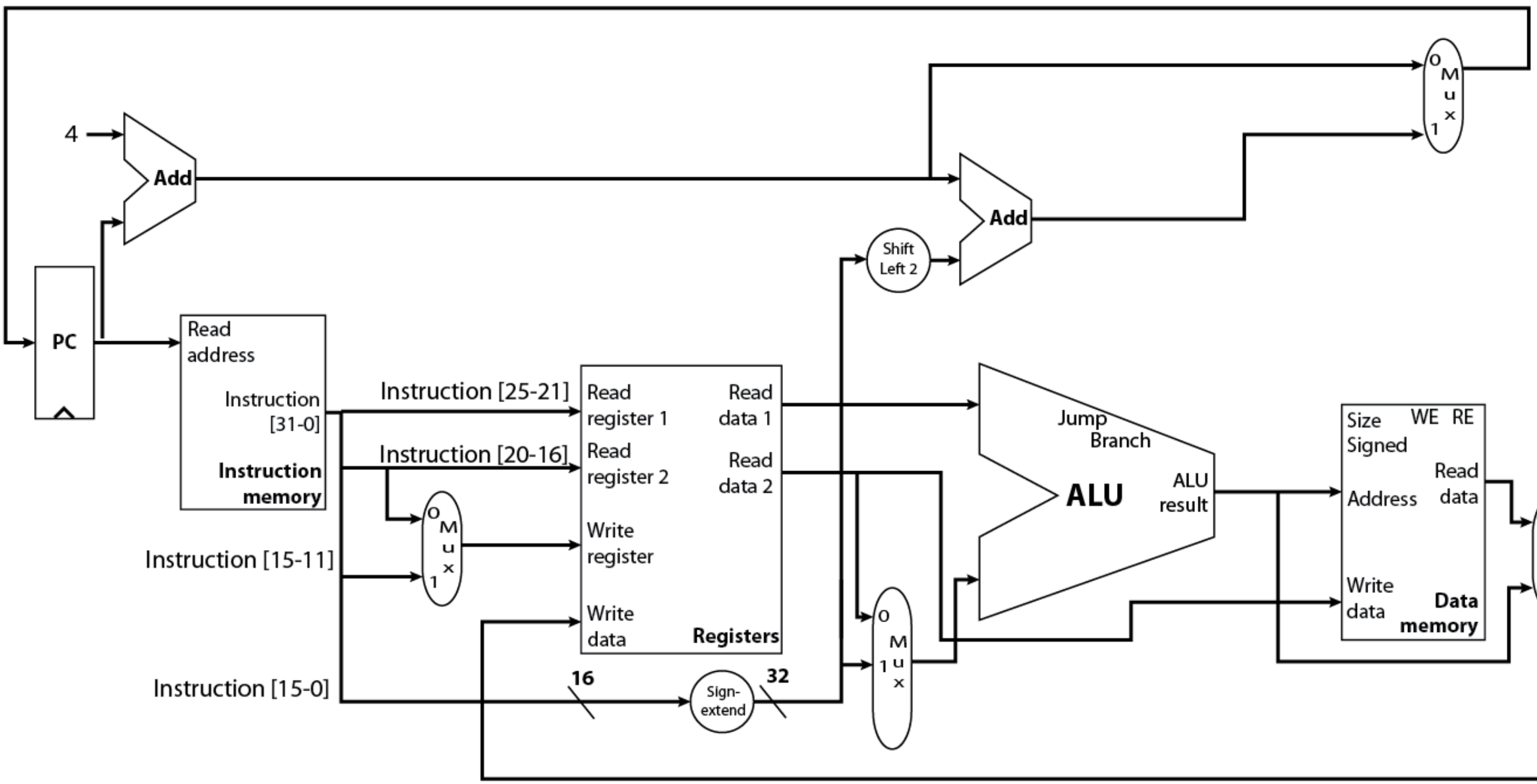
Cycle time = 18 ns

Slowest module (alu) is ~6ns

Our Hardware is Mostly Idle

Cycle time = 18 ns

Slowest module (alu) is ~6ns

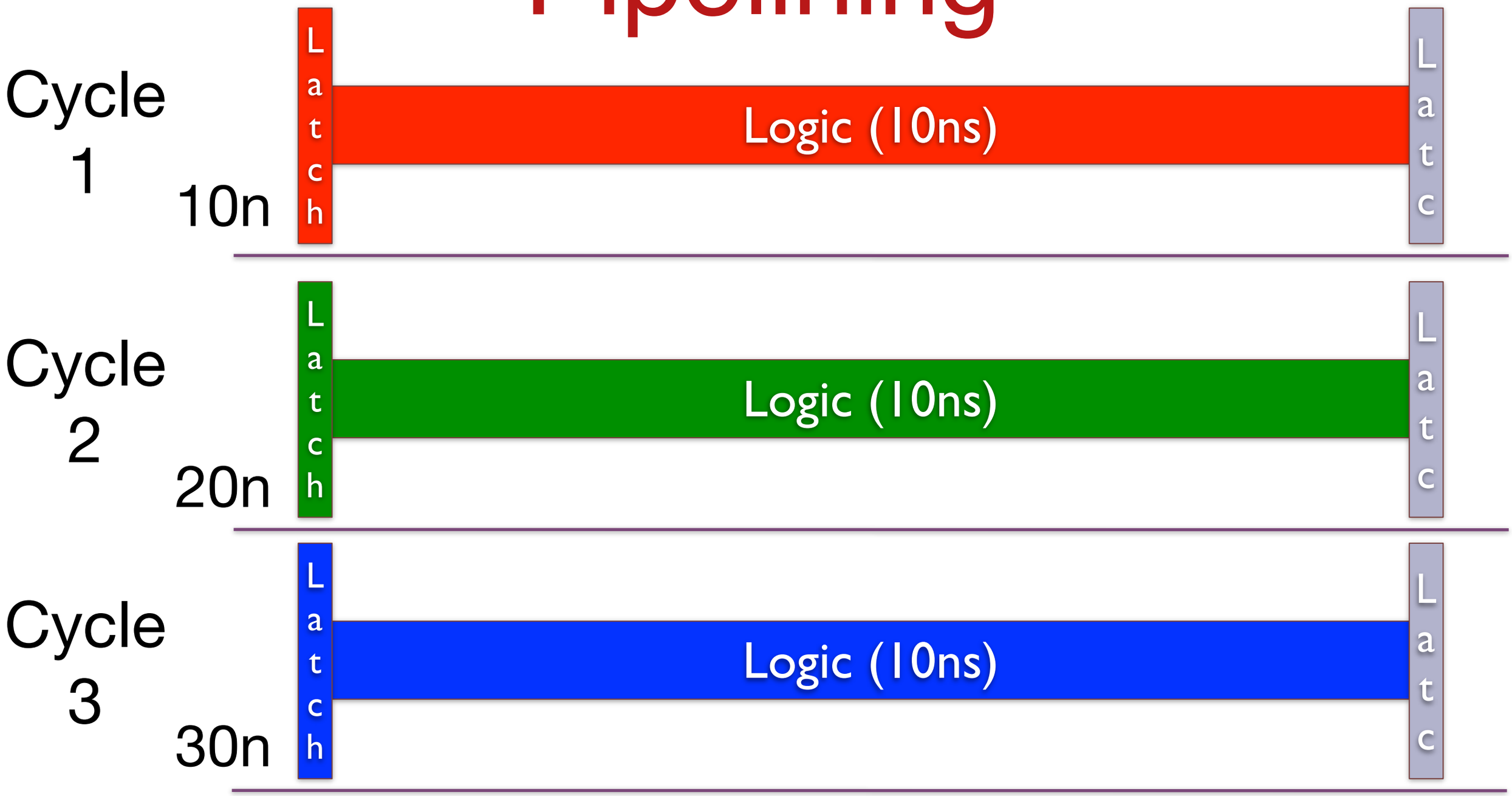


Processor Design in Two Acts

Act II: A pipelined CPU

Pipelining Review

Pipelining

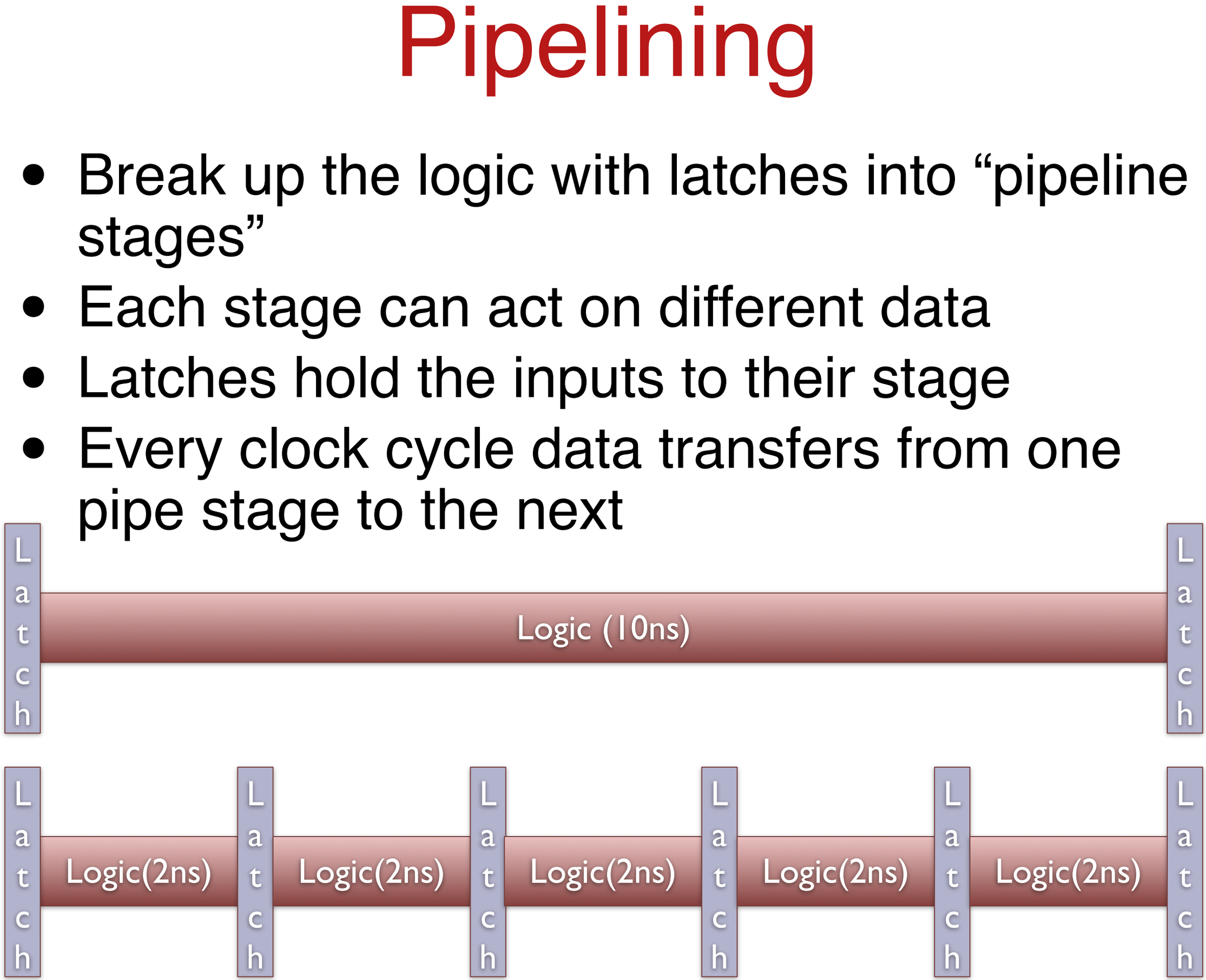


What's the throughput?

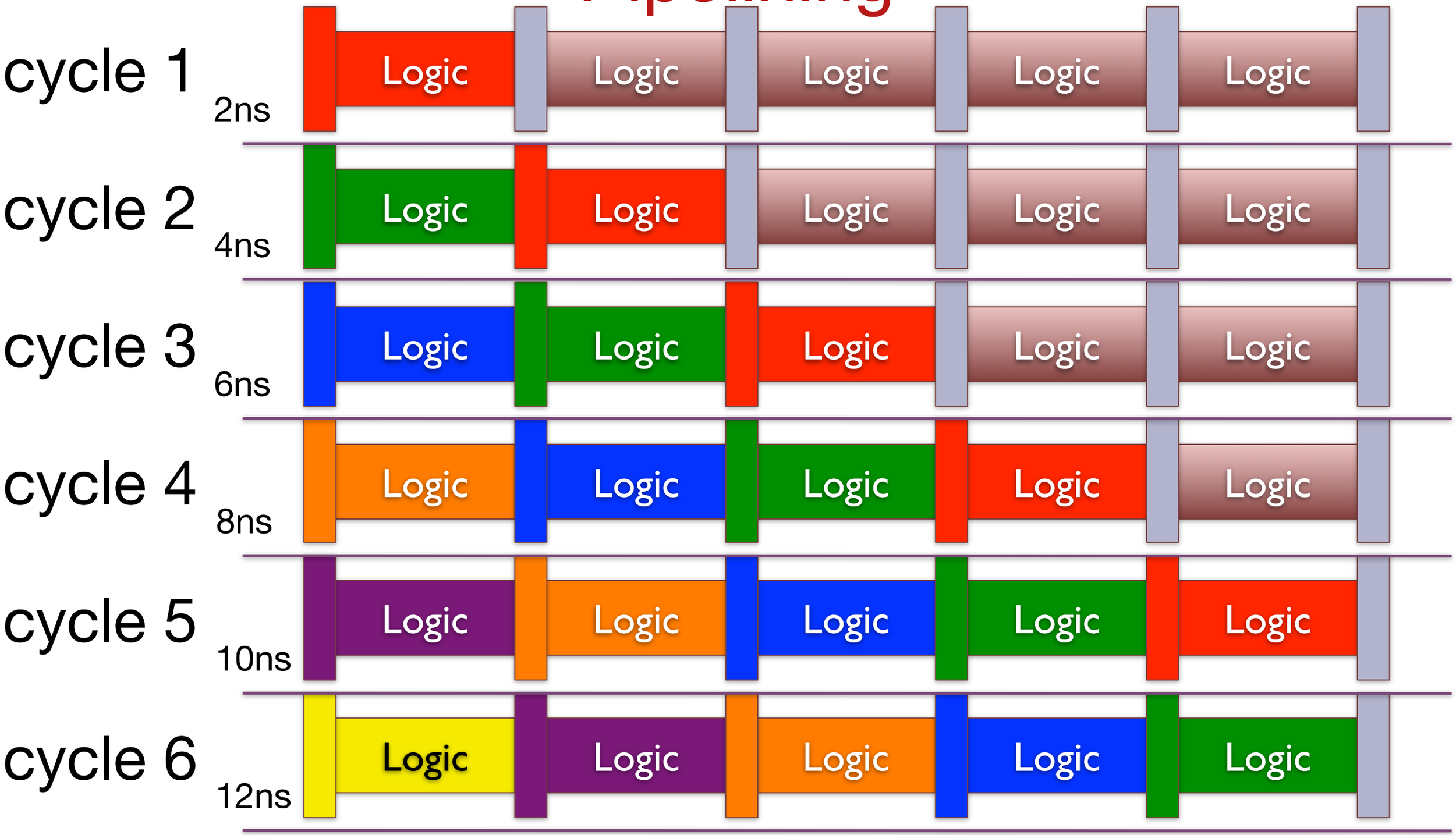
What's the latency for one unit of work?

Pipelining

- Break up the logic with latches into “pipeline stages”
- Each stage can act on different data
- Latches hold the inputs to their stage
- Every clock cycle data transfers from one pipe stage to the next



Pipelining

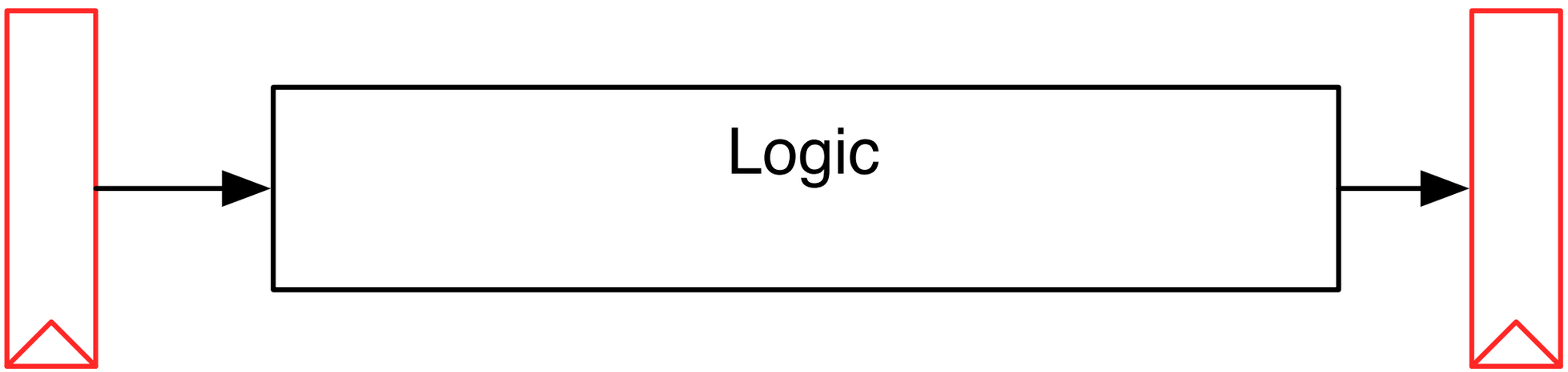


What's the latency for one unit of work?

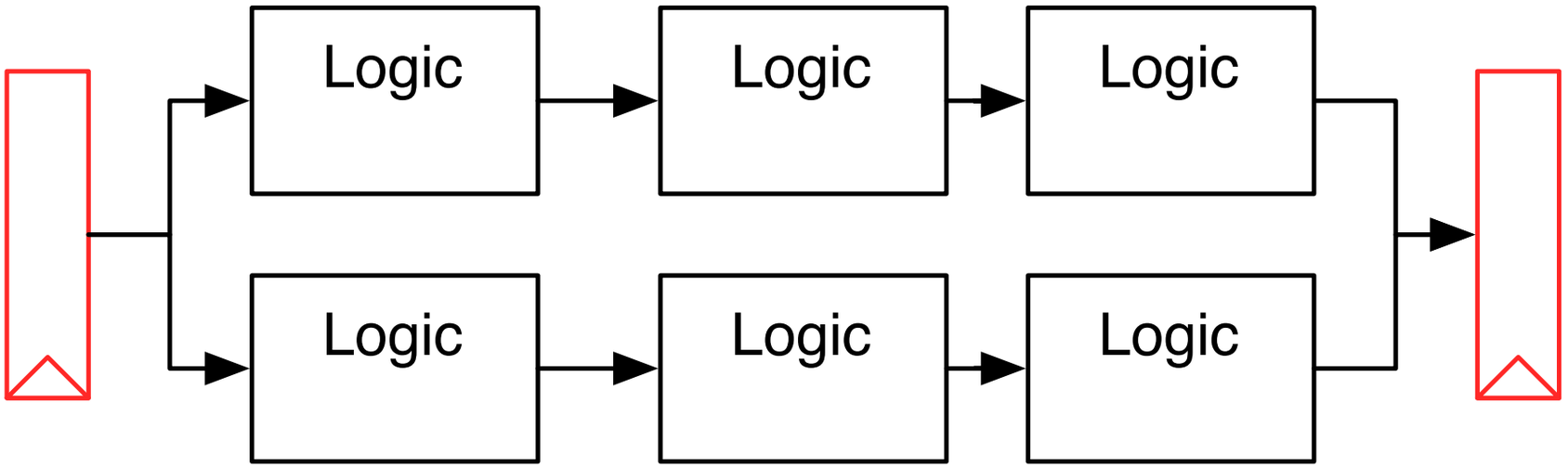
What's the throughput?

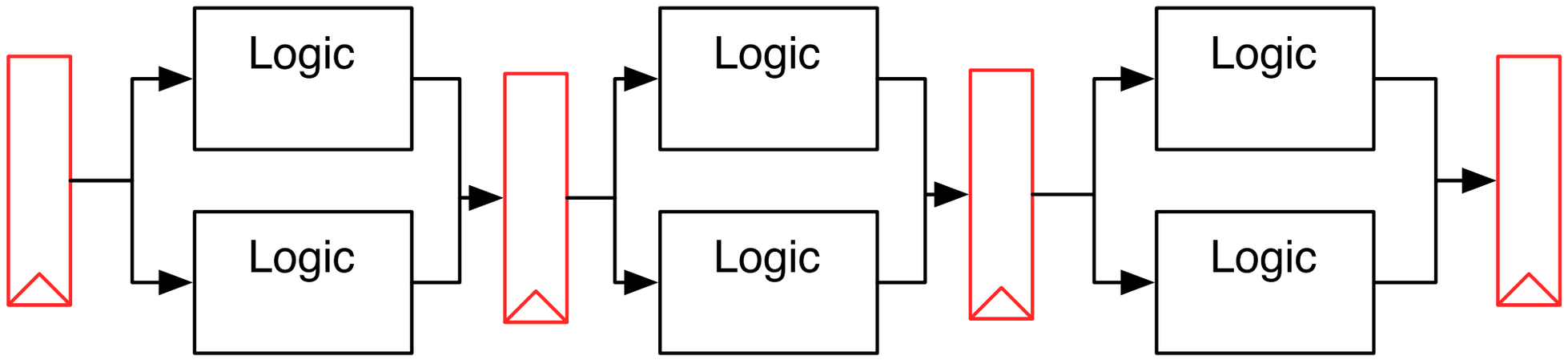
Critical path review

- Critical path is the longest possible delay between two registers in a design.
- The critical path sets the cycle time, since the cycle time must be long enough for a signal to traverse the critical path.
- Lengthening or shortening non-critical paths does not change performance
- Ideally, all paths are about the same length



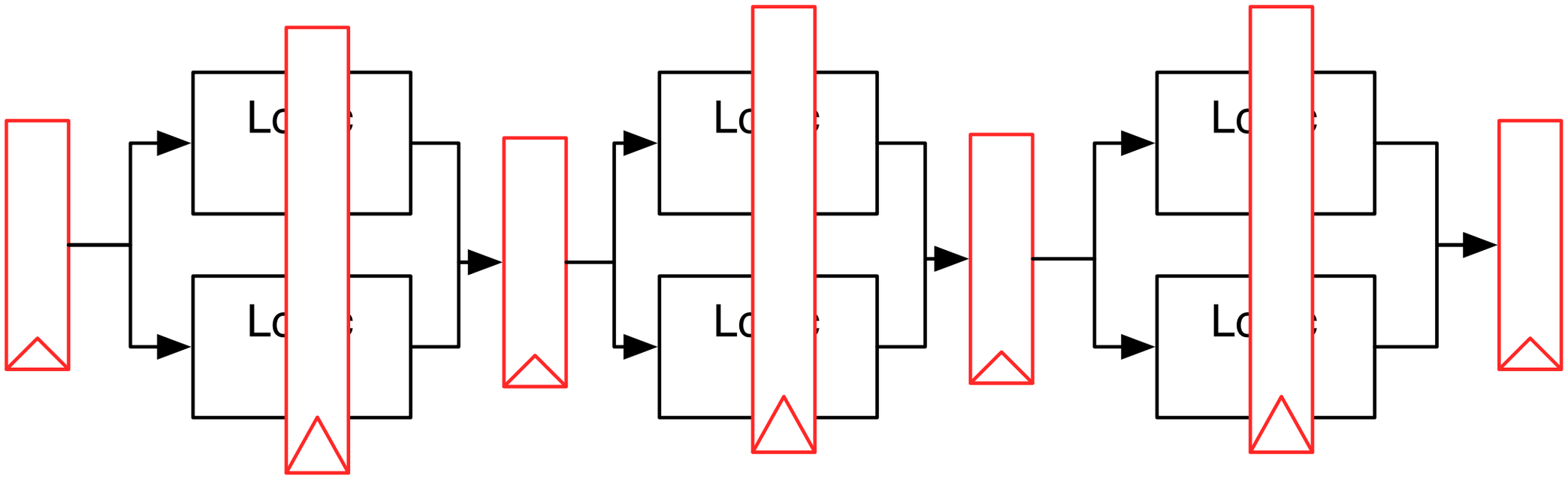
Pipelining and Logic





- Hopefully, critical path is now 1/3 of what it was

Limitations of Pipelining

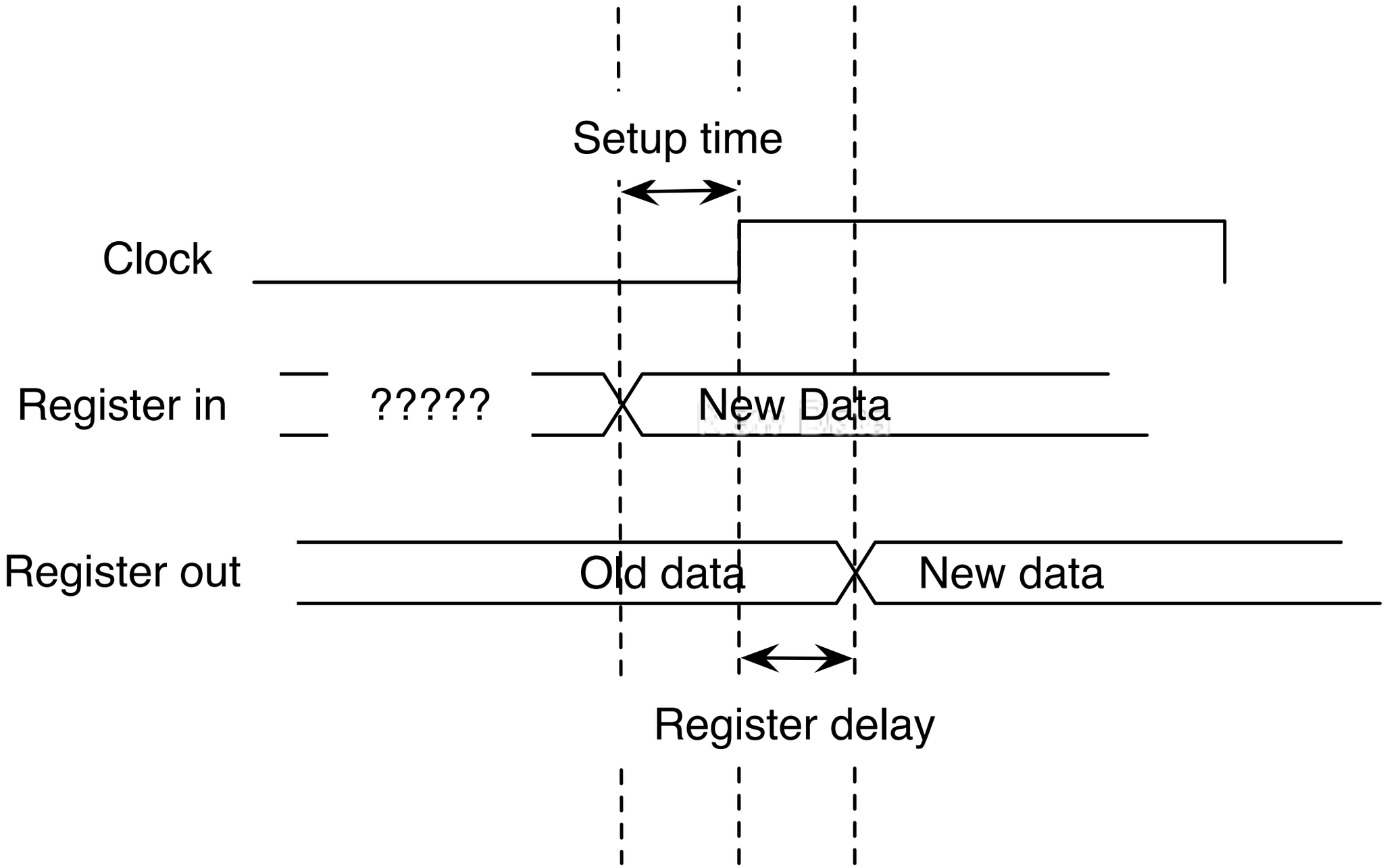


- You cannot pipeline forever
  - Some logic cannot be pipelined arbitrarily -- Memories
  - Some logic is inconvenient to pipeline.
  - How do you insert a register in the middle of an multiplier?
- Registers have a cost
  - They cost area -- choose “narrow points” in the logic
  - They cost energy -- latches don’t do any useful work
  - They cost time
    - Extra logic delay
    - Set-up and hold times.
- Pipelining may not affect the critical path as you expect

Pipelining Overhead

- Logic Delay (LD) -- How long does the logic take (i.e., the useful part)
- Set up time (ST) -- How long before the clock edge do the inputs to a register need be ready?
  - Relatively short -- 0.036 ns on our FPGAs
- Register delay (RD) -- Delay through the internals of the register.
  - Longer -- 1.5 ns for our FPGAs.
  - Much, much shorter for RAW CMOS.

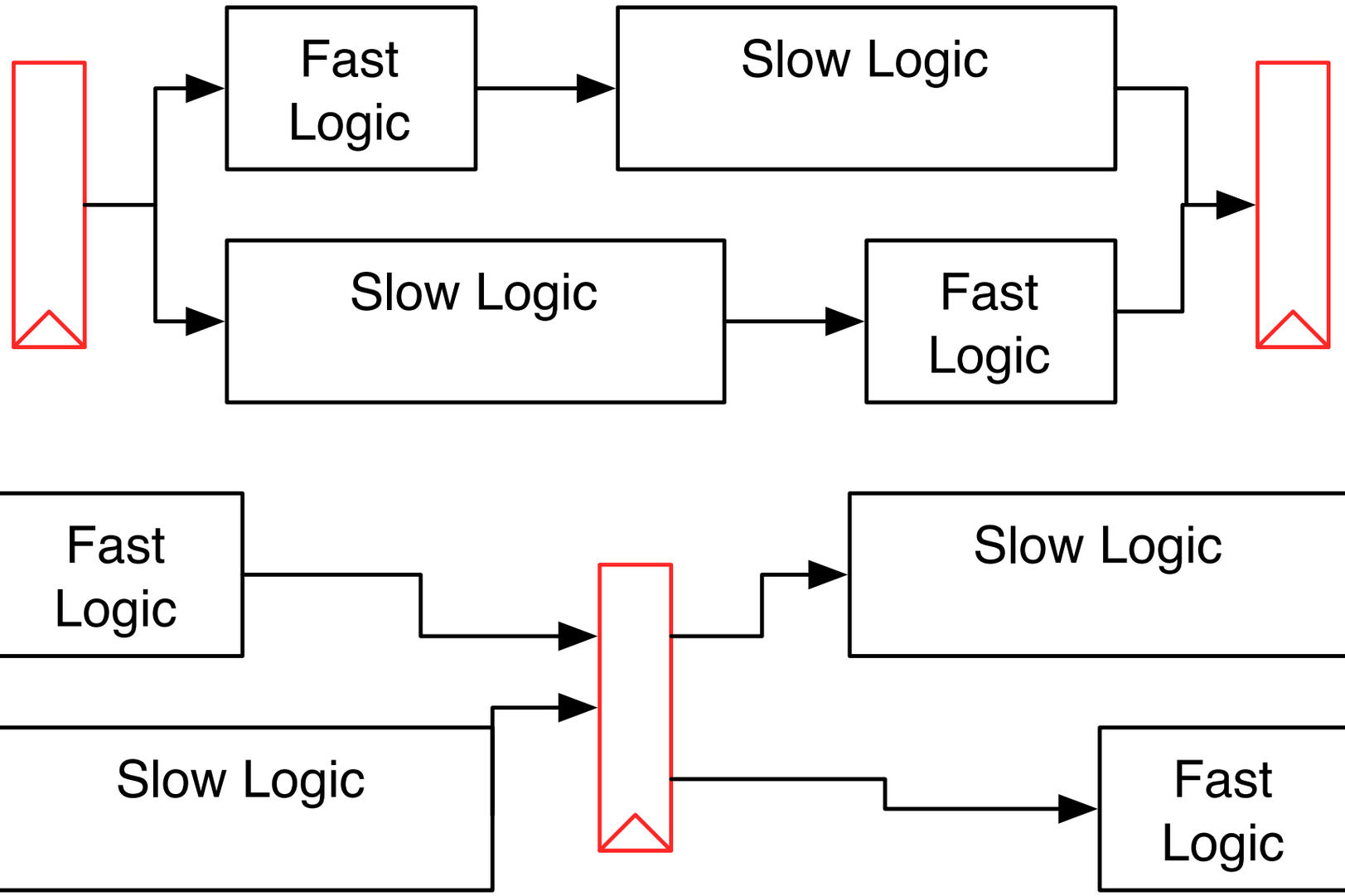
Pipelining Overhead



Pipelining Overhead

- Logic Delay (LD) -- How long does the logic take (i.e., the useful part)
- Set up time (ST) -- How long before the clock edge do the inputs to a register need be ready?
- Register delay (RD) -- Delay through the internals of the register.
- CT_{base} -- cycle time before pipelining
  - $CT_{base} = LD + ST + RD$.
- CT_{pipe} -- cycle time after pipelining *N* times
  - $CT_{pipe} = ST + RD + LD/N$
  - Total time = N*ST + N*RD + LD

Pipelining Difficulties



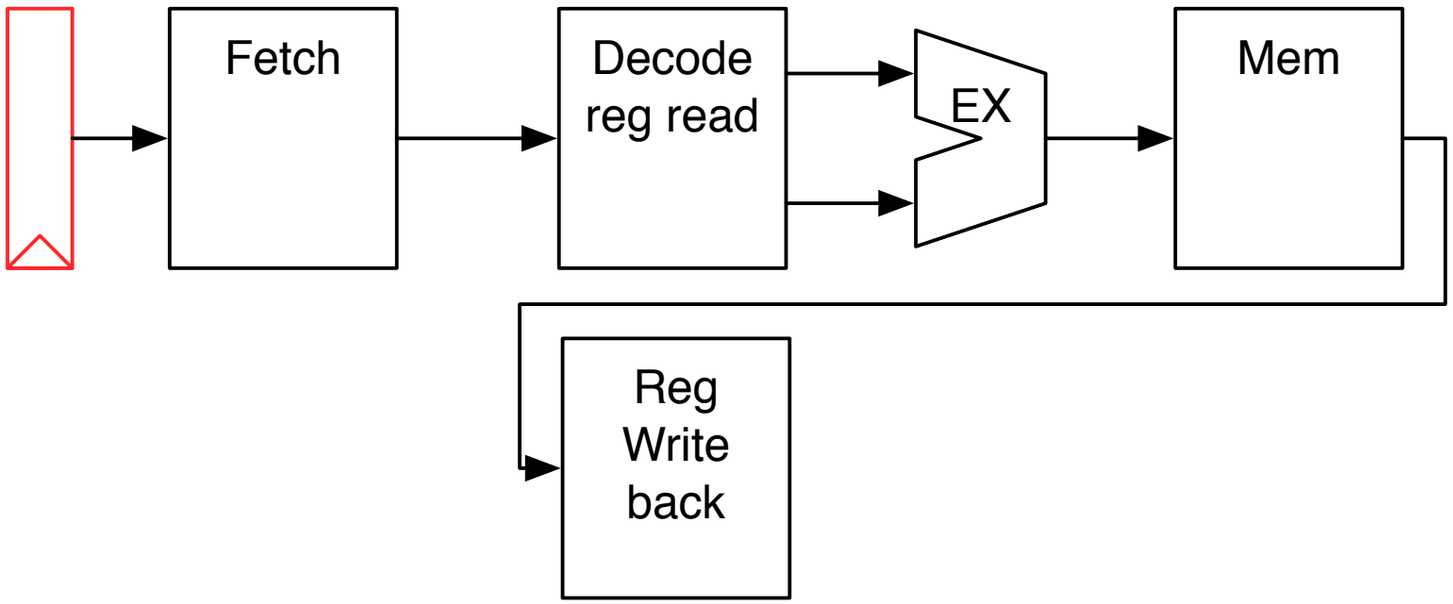
- You can’t always pipeline how you would like
- The critical path only went down “fast logic”

How to pipeline a processor

- Break each instruction into pieces
  - We'll base our break down on the basic algorithm for execution
    - Fetch
    - Decode
    - Collect arguments
    - Execute
    - Write back results
    - Compute next PC
- The "classic 5-stage MIPS pipeline"
  - Fetch -- read the instruction
  - Decode -- decode and read from the register file
  - Execute -- Perform arithmetic ops and address calculations
  - Memory -- access data memory.
  - Write back-- Store results in the register file.

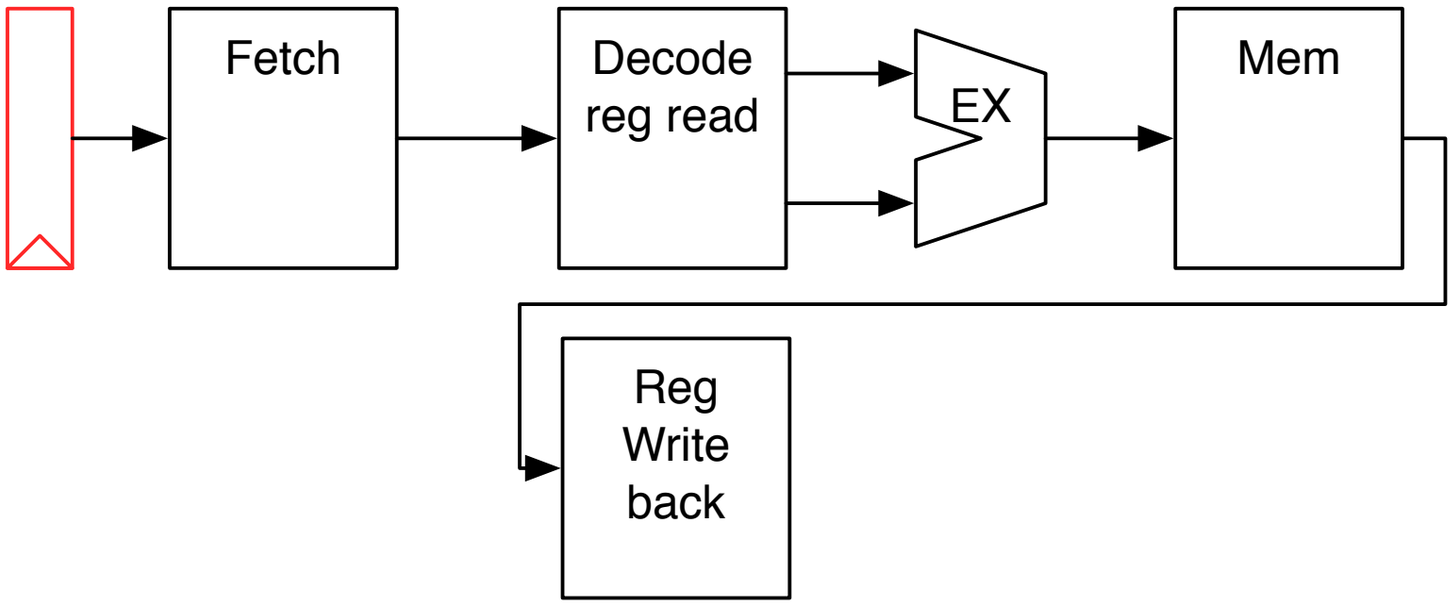
Pipelining a processor

Pipelining a processor

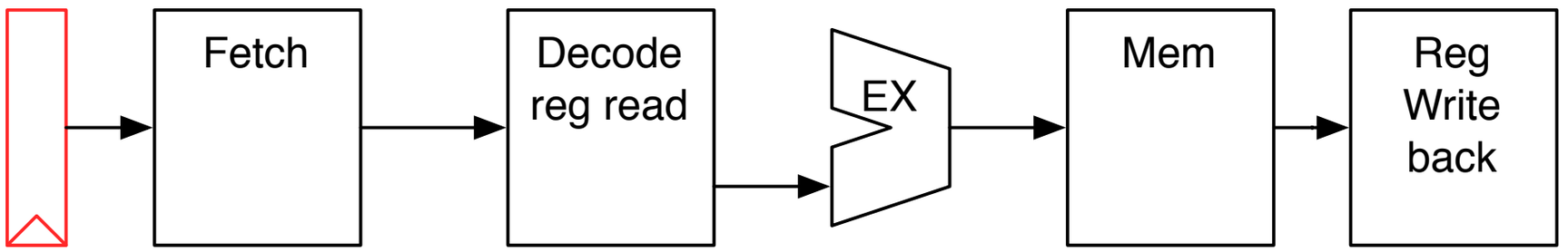


Reality

Pipelining a processor

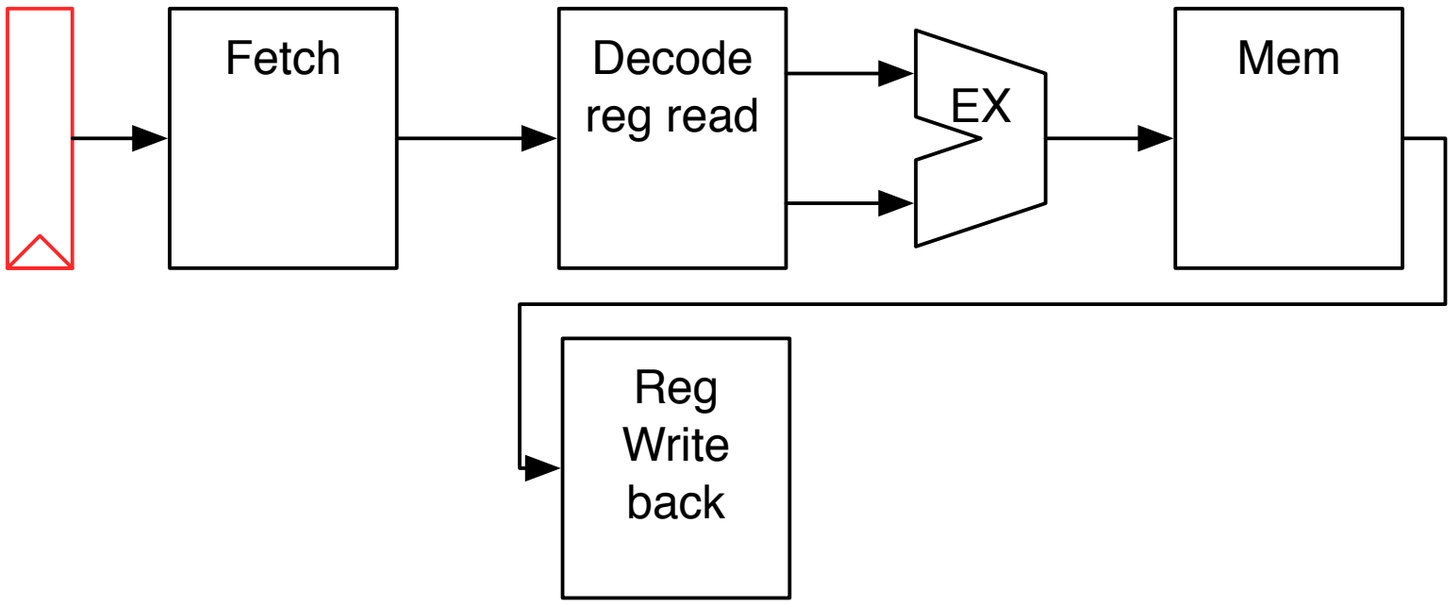


Reality

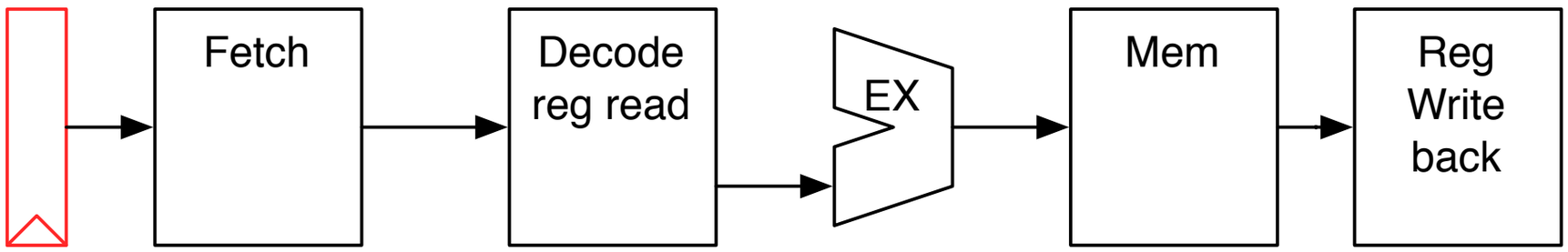


Easier to draw

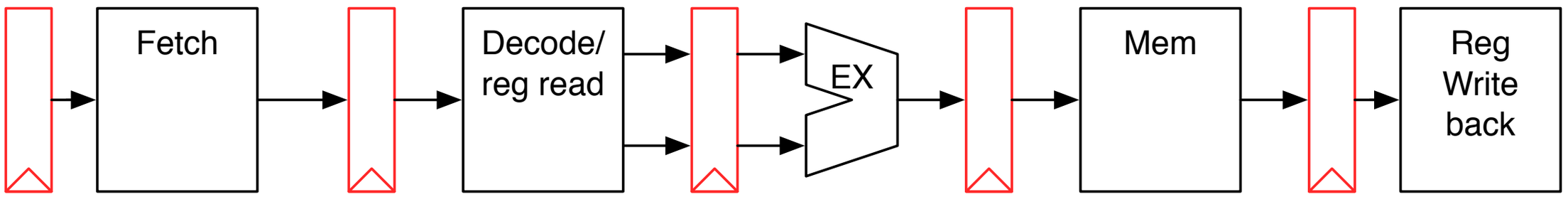
Pipelining a processor



Reality

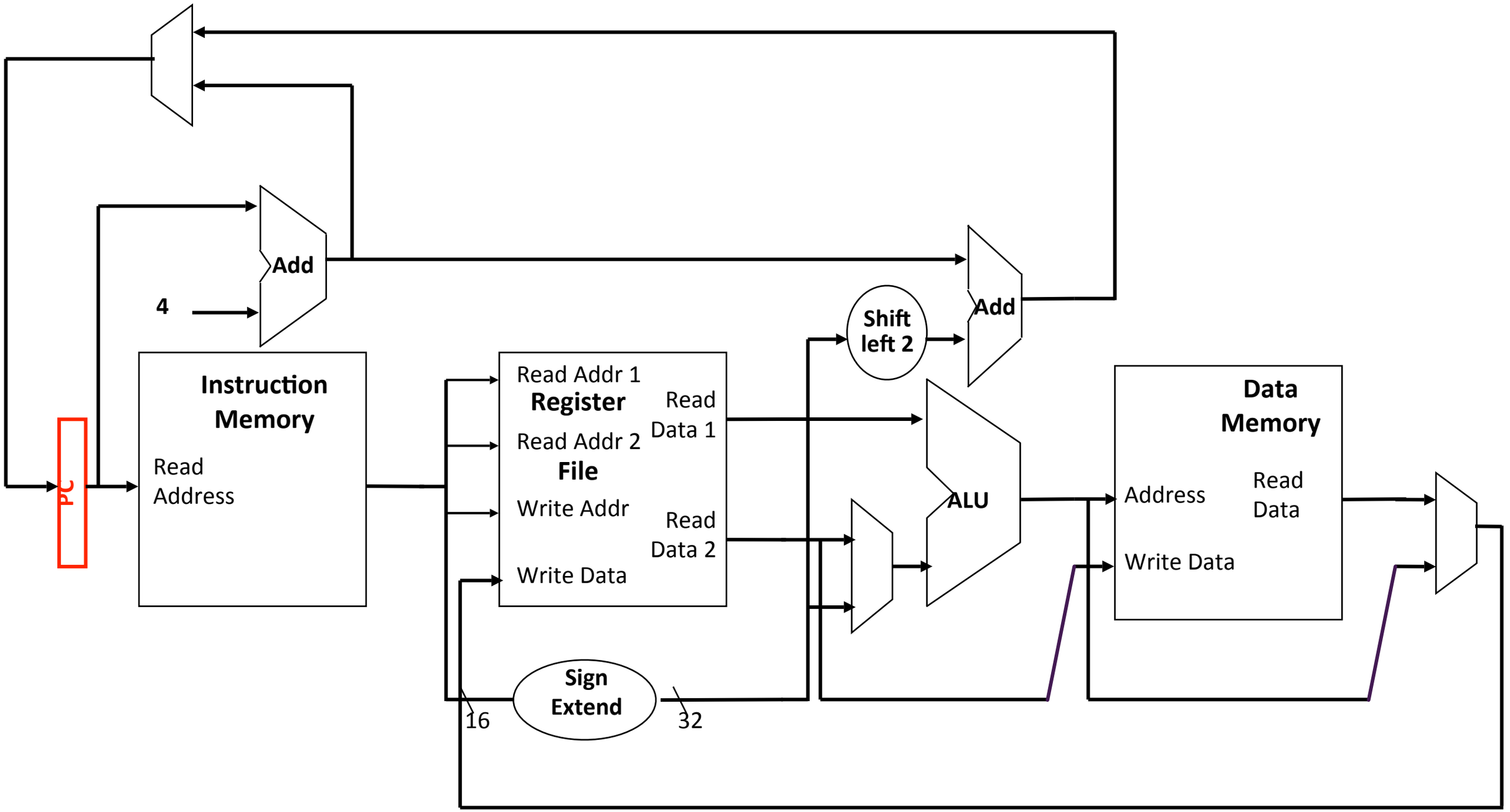


Easier to draw

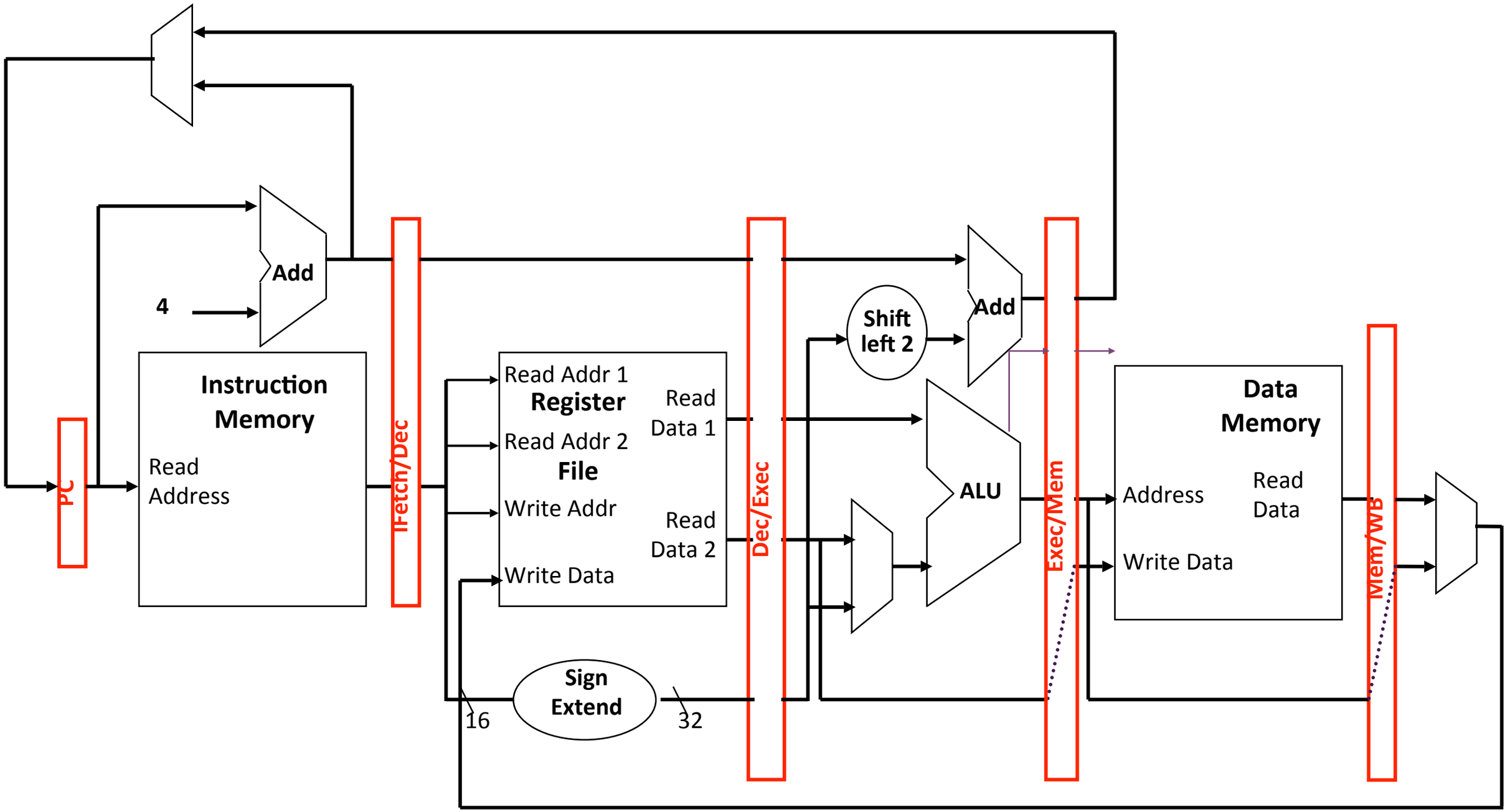


Pipelined

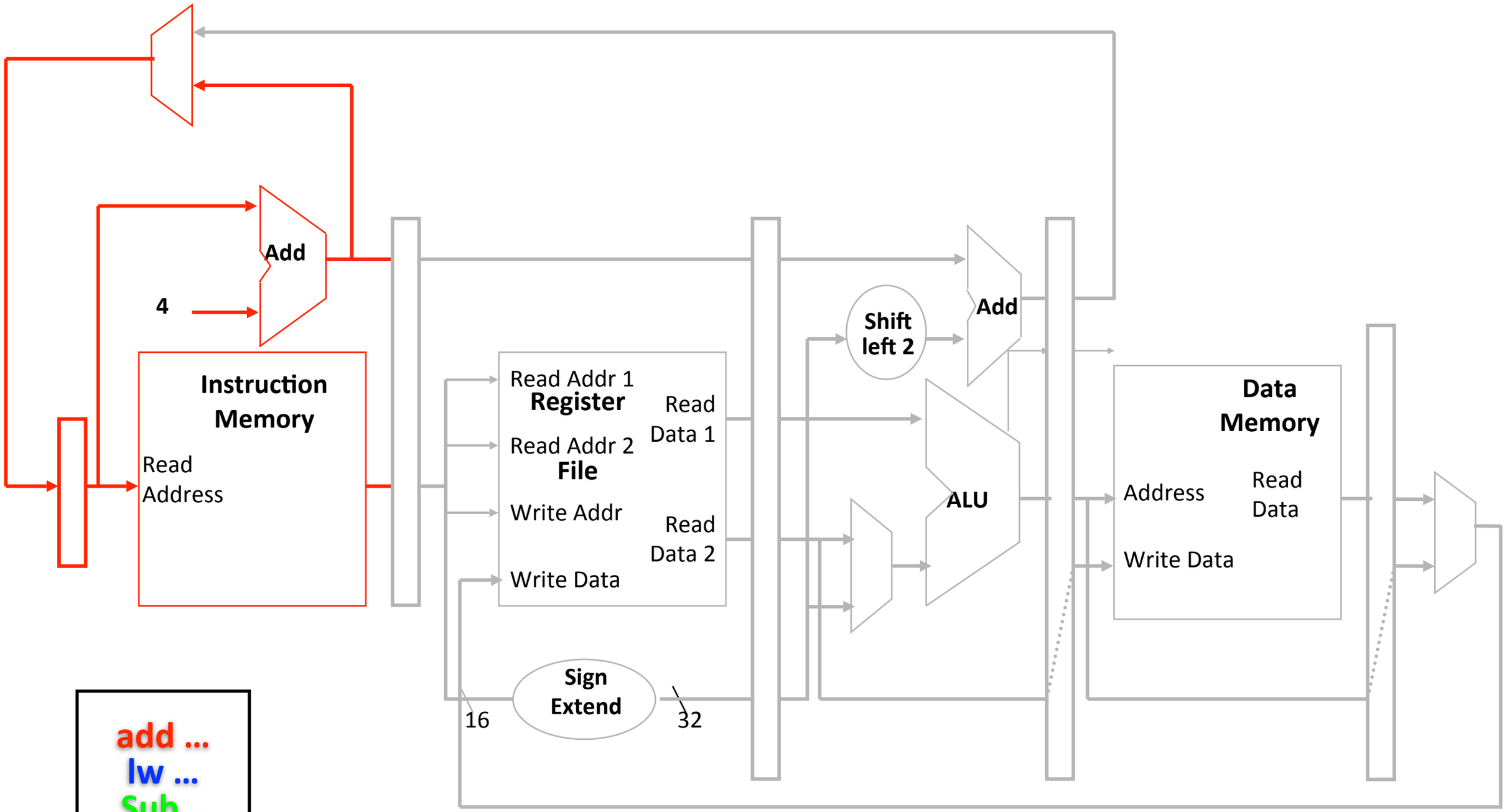
Pipelined Datapath



Pipelined Datapath

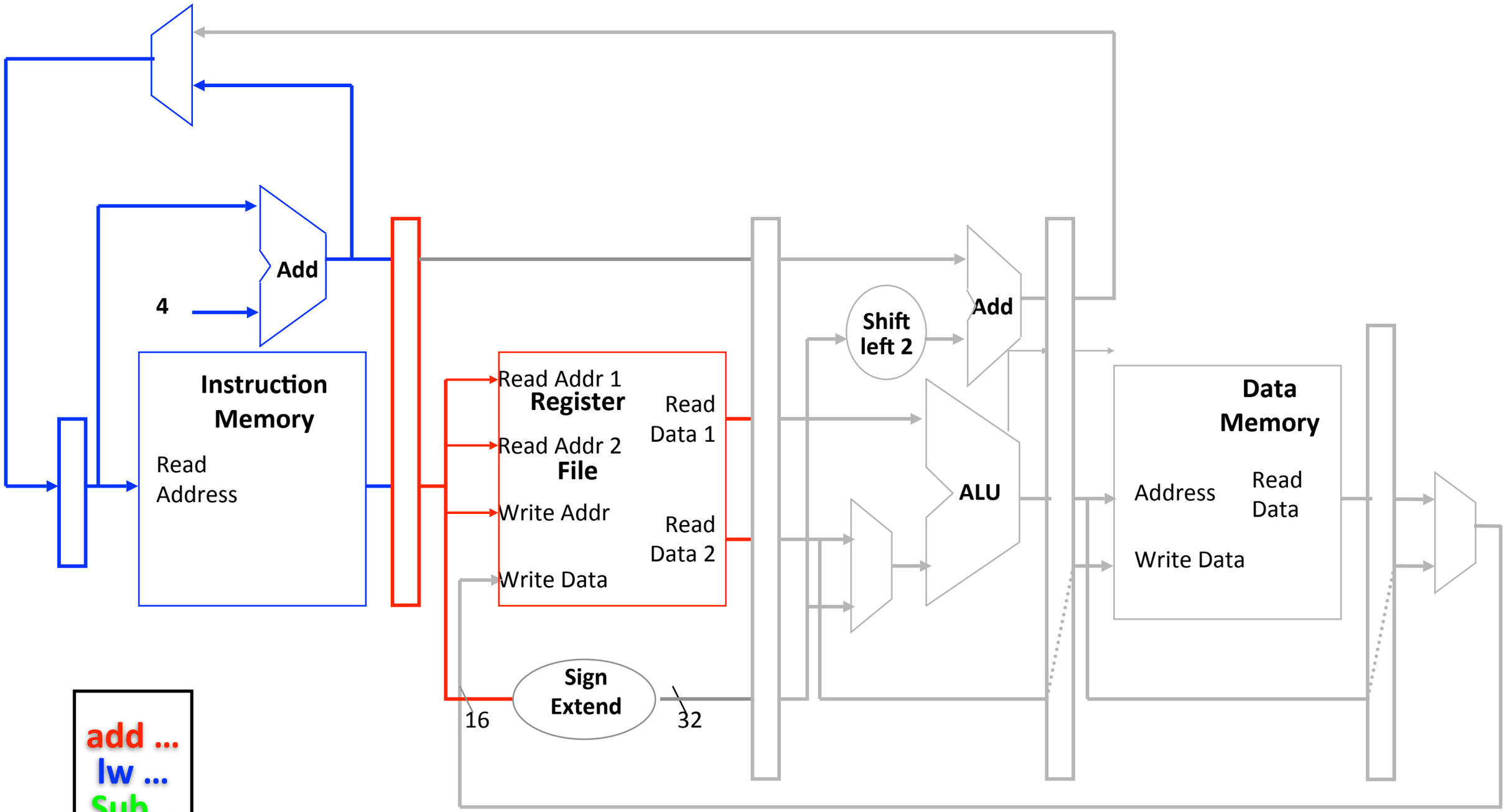


Pipelined Datapath



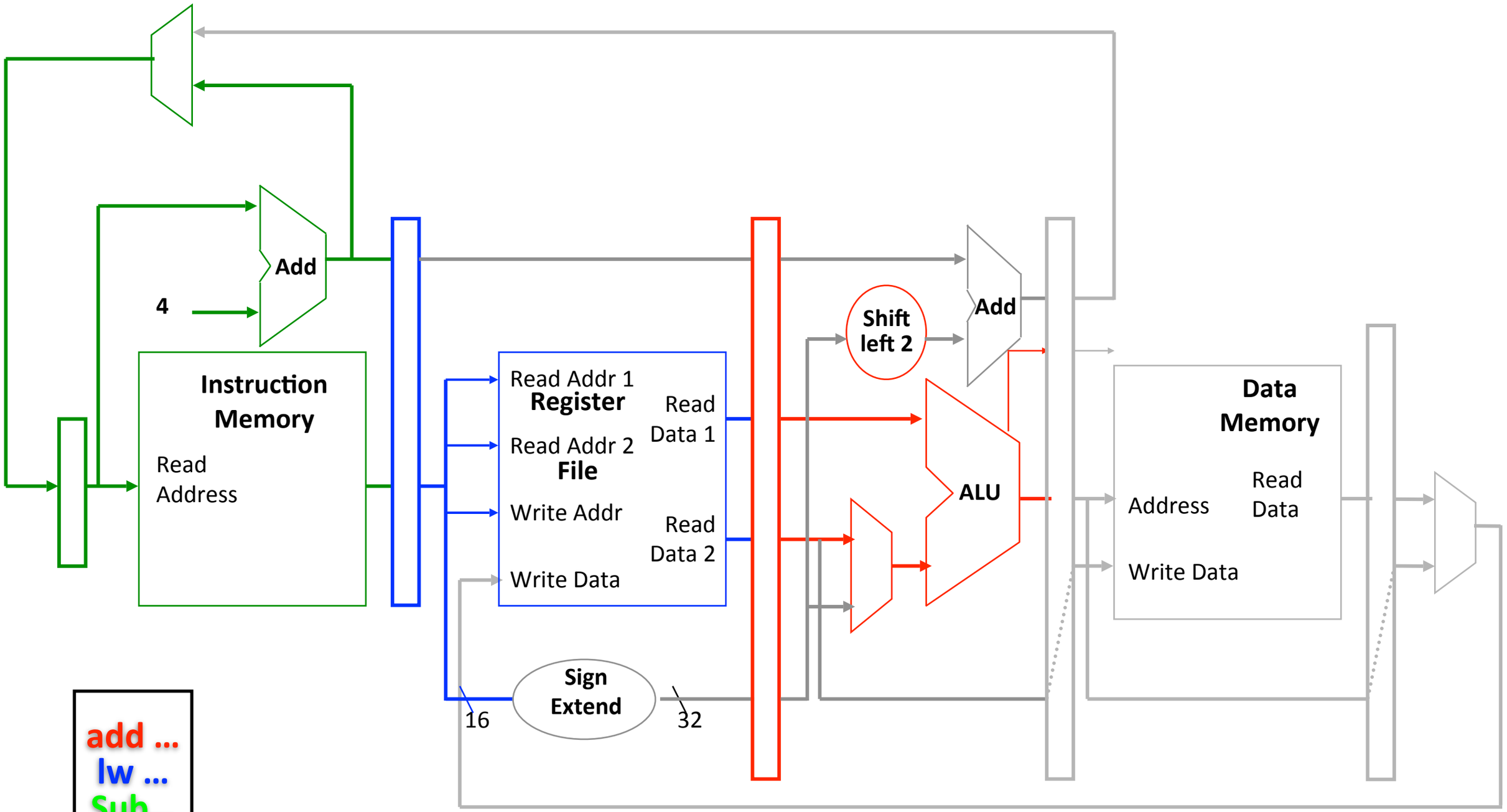
add …
lw …
Sub…
Sub ….
Add …
Add …

Pipelined Datapath



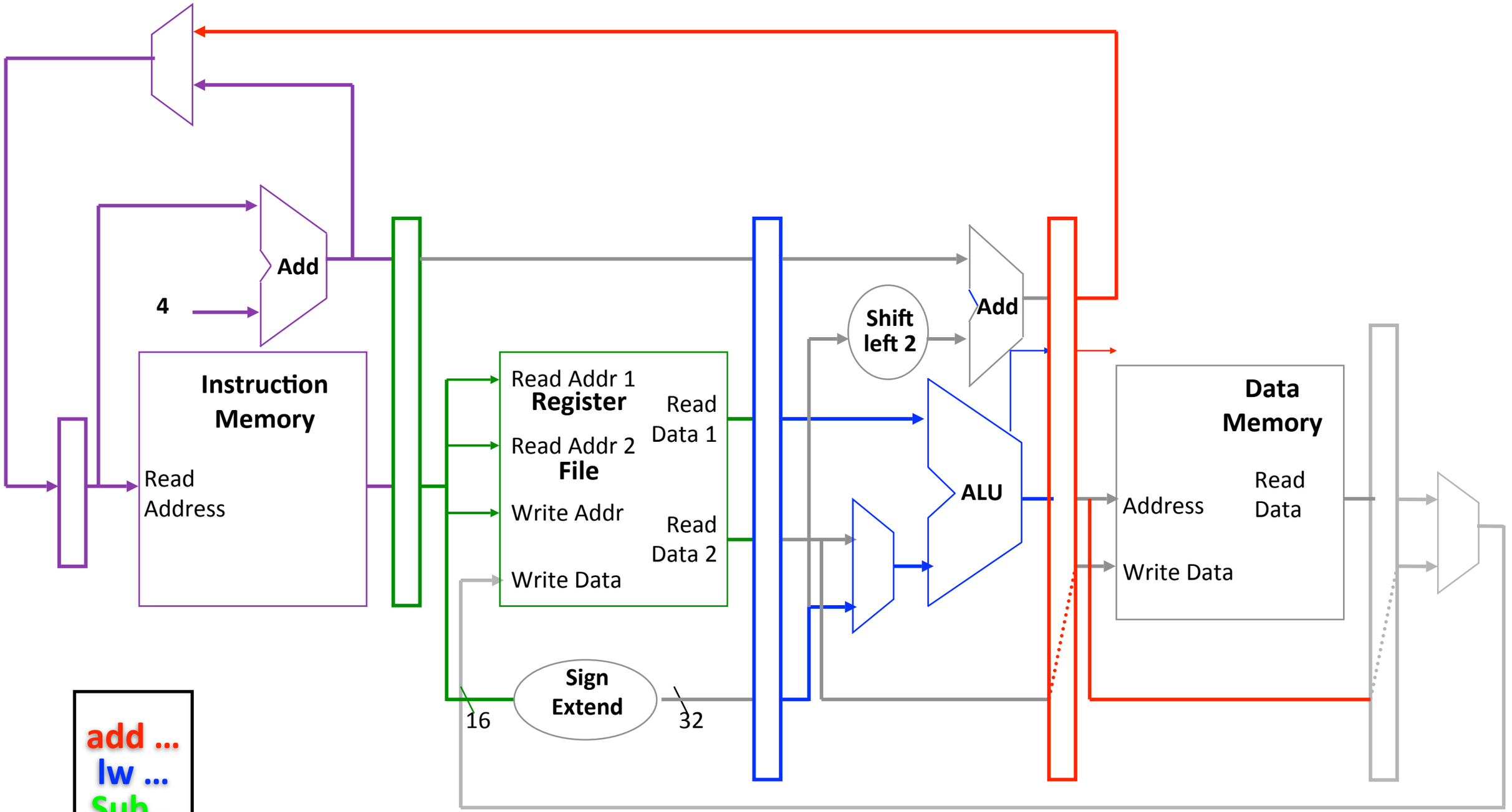
add ...
lw ...
Sub...
Sub ....
Add ...
Add ...

Pipelined Datapath



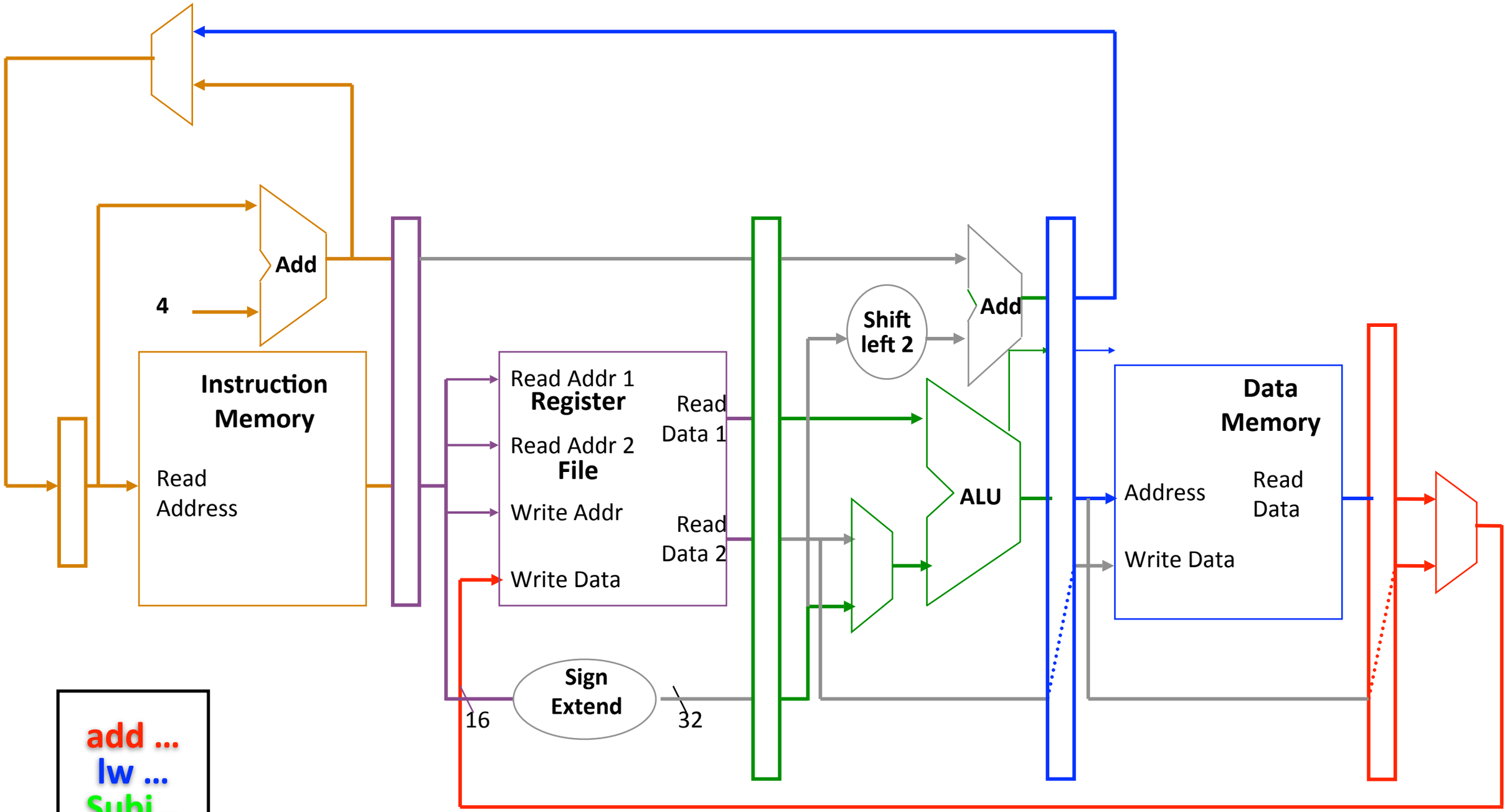
add …
lw …
Sub…
Sub ….
Add …
Add …

Pipelined Datapath



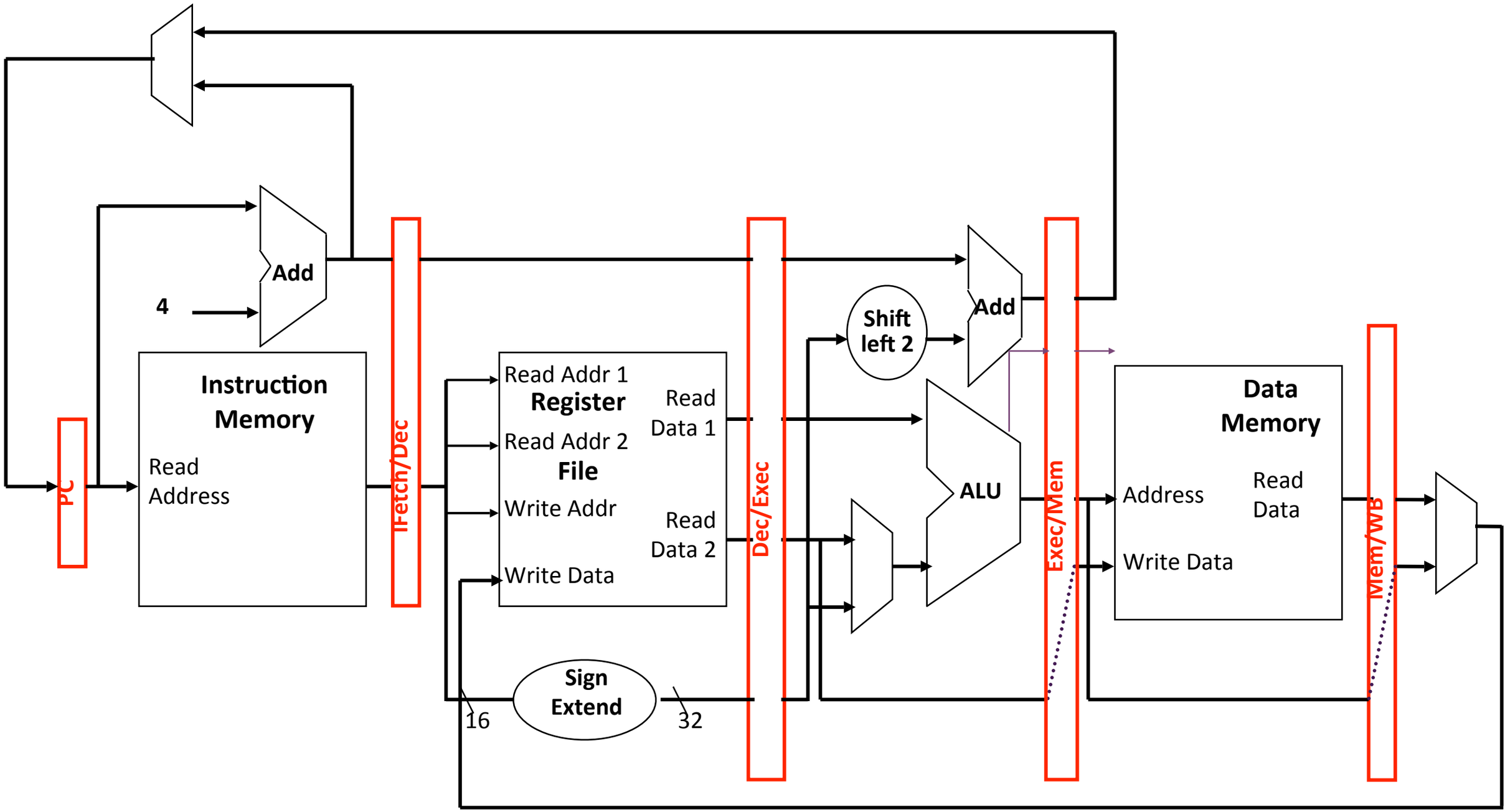
add …
lw …
Sub…
Sub ….
Add …
Add …

Pipelined Datapath

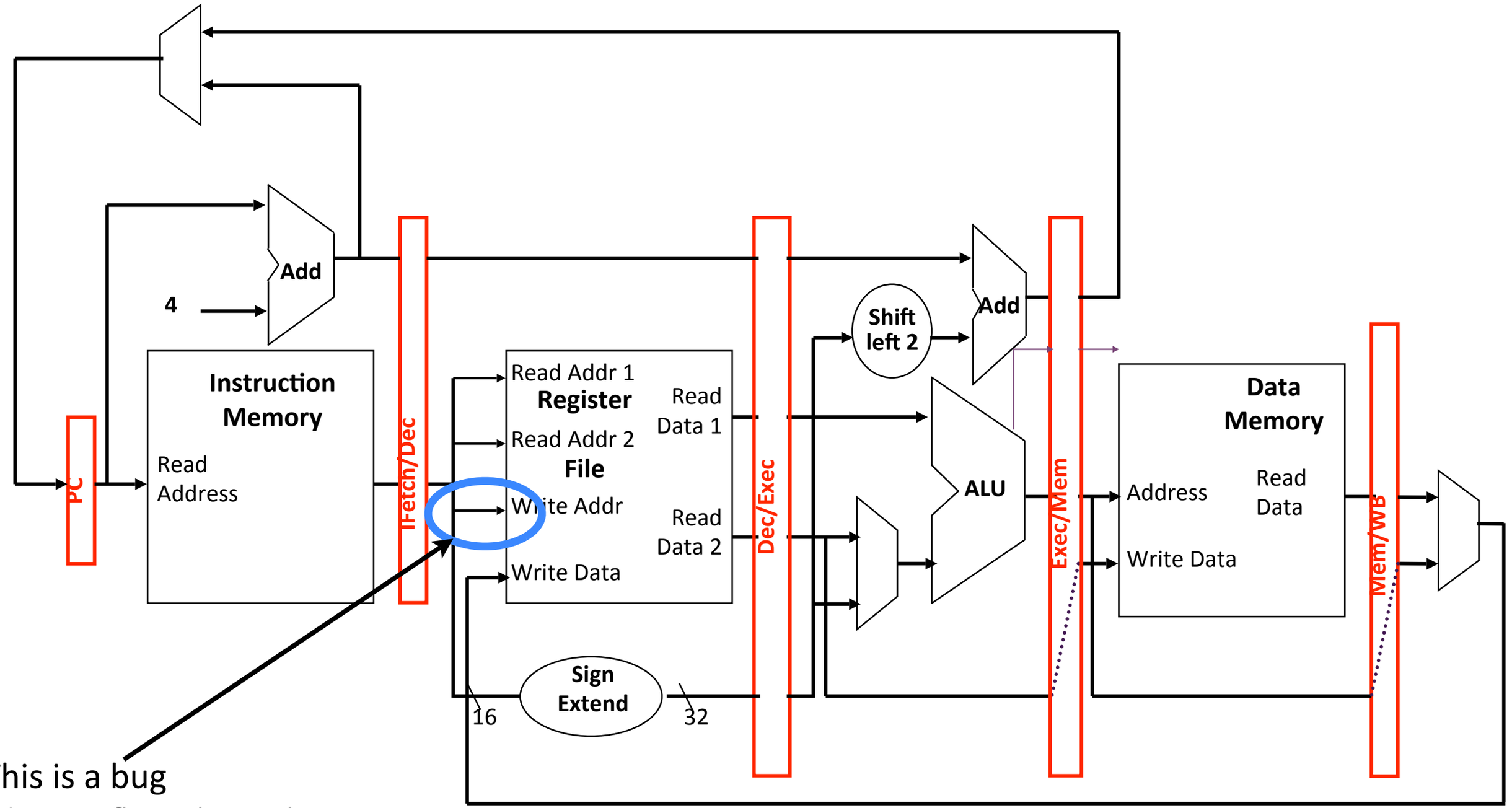


add …
lw …
Subi…
Sub ….
Add …
Add …

Pipelined Datapath

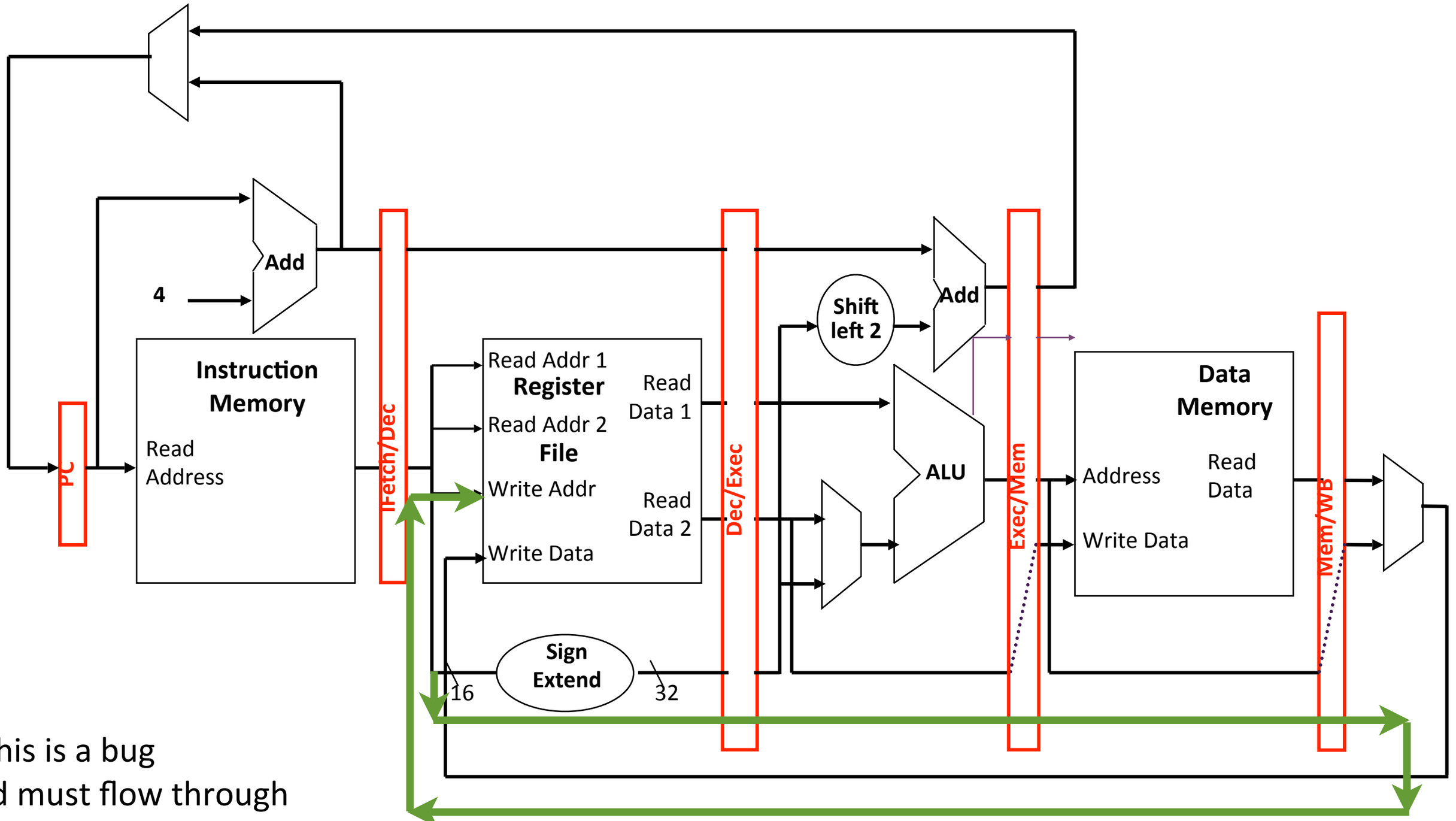


Pipelined Datapath



This is a bug
rd must flow through
the pipeline with the instruction.
This signal needs to come from the WB stage.

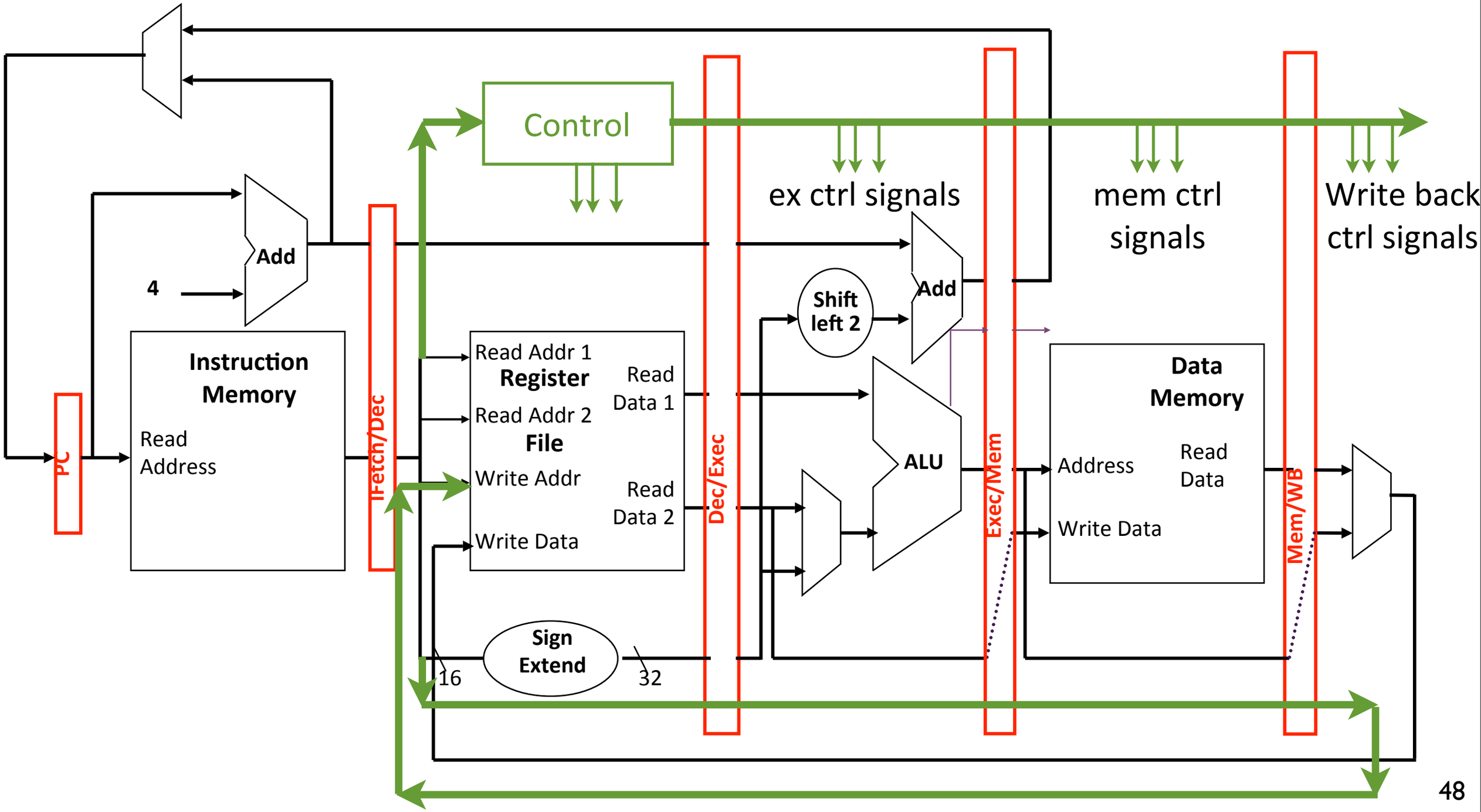
Pipelined Datapath



This is a bug
rd must flow through
the pipeline with the instruction.
This signal needs to come from the WB stage.

Pipelined Control

- Control lives in decode stage, signals flow down the pipe with the instructions they correspond to

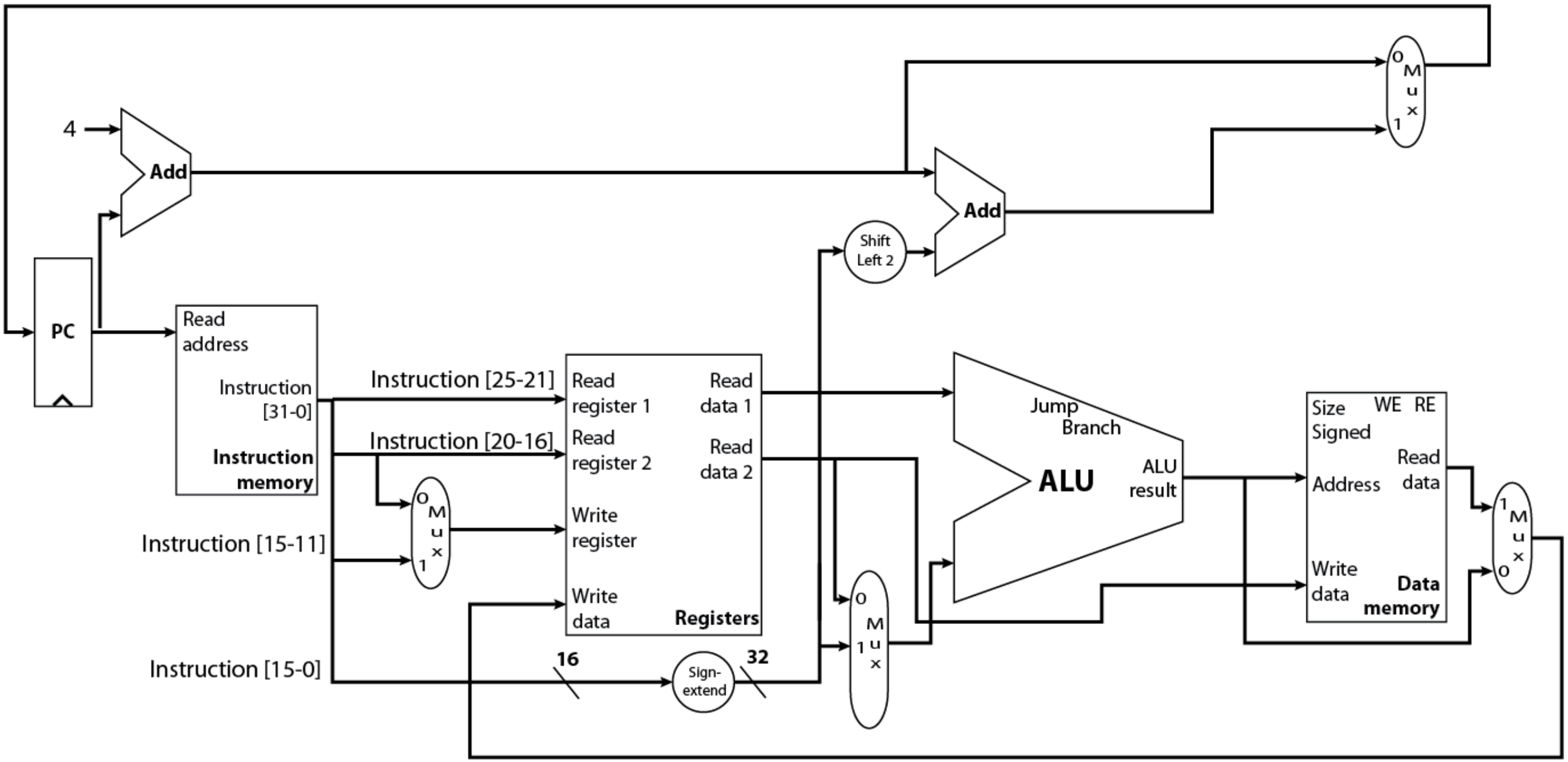


Impact of Pipelining

- L = IC * CPI * CT
- Break the processor into P pipe stages
  - $CT_{new} = CT/P$
  - $CPI_{new} = CPI_{old}$
    - CPI is an average: Cycles/instructions
    - The latency of *one instruction* is P cycles
    - The average CPI = 1
  - $IC_{new} = IC_{old}$
- Total speedup should be 5x!
  - Except for the overhead of the pipeline registers
  - And the realities of logic design...

Pipelining Inaction

Pipelining Inaction



Total	Incr	RF	Type	Fanout	Location	Element
18.667	15.810					data path
3.134	0.277		uTco	1	LCFF_X21_Y16_N25	inst_rom:rom|out[5]
3.134	0.000	FF	CELL	23	LCFF_X21_Y16_N25	rom|out[5]|regout
3.754	0.620	FF	IC	1	LCCOMB_X20_Y16_N20	ctrl|Mux3~1|datad
3.931	0.177	FR	CELL	29	LCCOMB_X20_Y16_N20	ctrl|Mux3~1|combout
4.877	0.946	RR	IC	1	LCCOMB_X19_Y12_N4	muxImmediateMode|Mux25~0|datad
5.055	0.178	RR	CELL	5	LCCOMB_X19_Y12_N4	muxImmediateMode|Mux25~0|combout
5.643	0.588	RR	IC	1	LCCOMB_X18_Y12_N20	theAlu|AdderInputB[6]~29|dataa
6.188	0.545	RR	CELL	1	LCCOMB_X18_Y12_N20	theAlu|AdderInputB[6]~29|combout
7.094	0.906	RR	IC	2	LCCOMB_X19_Y16_N30	theAlu|Add0~14|datab
7.689	0.595	RF	CELL	1	LCCOMB_X19_Y16_N30	theAlu|Add0~14|cout
7.689	0.000	FF	IC	2	LCCOMB_X19_Y15_N0	theAlu|Add0~16|cin
7.769	0.080	FR	CELL	1	LCCOMB_X19_Y15_N0	theAlu|Add0~16|cout
7.769	0.000	RR	IC	2	LCCOMB_X19_Y15_N2	theAlu|Add0~18|cin
7.849	0.080	RF	CELL	1	LCCOMB_X19_Y15_N2	theAlu|Add0~18|cout
7.849	0.000	FF	IC	2	LCCOMB_X19_Y15_N4	theAlu|Add0~20|cin
7.929	0.080	FR	CELL	1	LCCOMB_X19_Y15_N4	theAlu|Add0~20|cout
7.929	0.000	RR	IC	2	LCCOMB_X19_Y15_N6	theAlu|Add0~22|cin
8.009	0.080	RF	CELL	1	LCCOMB_X19_Y15_N6	theAlu|Add0~22|cout
8.009	0.000	FF	IC	2	LCCOMB_X19_Y15_N8	theAlu|Add0~24|cin
8.089	0.080	FR	CELL	1	LCCOMB_X19_Y15_N8	theAlu|Add0~24|cout
8.089	0.000	RR	IC	2	LCCOMB_X19_Y15_N10	theAlu|Add0~26|cin
8.169	0.080	RF	CELL	1	LCCOMB_X19_Y15_N10	theAlu|Add0~26|cout
8.169	0.000	FF	IC	2	LCCOMB_X19_Y15_N12	theAlu|Add0~28|cin
8.249	0.080	FR	CELL	1	LCCOMB_X19_Y15_N12	theAlu|Add0~28|cout
8.249	0.000	RR	IC	2	LCCOMB_X19_Y15_N14	theAlu|Add0~30|cin
8.423	0.174	RF	CELL	1	LCCOMB_X19_Y15_N14	theAlu|Add0~30|cout
8.423	0.000	FF	IC	2	LCCOMB_X19_Y15_N16	theAlu|Add0~32|cin
8.503	0.080	FR	CELL	1	LCCOMB_X19_Y15_N16	theAlu|Add0~32|cout
8.503	0.000	RR	IC	2	LCCOMB_X19_Y15_N18	theAlu|Add0~34|cin
8.583	0.080	RF	CELL	1	LCCOMB_X19_Y15_N18	theAlu|Add0~34|cout
8.583	0.000	FF	IC	2	LCCOMB_X19_Y15_N20	theAlu|Add0~36|cin
8.663	0.080	FR	CELL	1	LCCOMB_X19_Y15_N20	theAlu|Add0~36|cout
8.663	0.000	RR	IC	2	LCCOMB_X19_Y15_N22	theAlu|Add0~38|cin
8.743	0.080	RF	CELL	1	LCCOMB_X19_Y15_N22	theAlu|Add0~38|cout
8.743	0.000	FF	IC	2	LCCOMB_X19_Y15_N24	theAlu|Add0~40|cin
8.823	0.080	FR	CELL	1	LCCOMB_X19_Y15_N24	theAlu|Add0~40|cout
8.823	0.000	RR	IC	2	LCCOMB_X19_Y15_N26	theAlu|Add0~42|cin
8.903	0.080	RF	CELL	1	LCCOMB_X19_Y15_N26	theAlu|Add0~42|cout

Total	Incr	RF	Type	Fanout	Location	Element
18.667	15.810					data path
3.134	0.277		uTco	1	LCFF_X21_Y16_N25	inst_rom:rom|out[5]
3.134	0.000	FF	CELL	23	LCFF_X21_Y16_N25	rom|out[5]|regout
3.754	0.620	FF	IC	1	LCCOMB_X20_Y16_N20	ctrl|Mux3~1|datad
3.931	0.177	FR	CELL	29	LCCOMB_X20_Y16_N20	ctrl|Mux3~1|combout
4.877	0.946	RR	IC	1	LCCOMB_X19_Y12_N4	muxImmediateMode|Mux25~0|datad
5.055	0.178	RR	CELL	5	LCCOMB_X19_Y12_N4	muxImmediateMode|Mux25~0|combout
5.643	0.588	RR	IC	1	LCCOMB_X18_Y12_N20	theAlu|AdderInputB[6]~29|dataa
6.188	0.545	RR	CELL	1	LCCOMB_X18_Y12_N20	theAlu|AdderInputB[6]~29|combout
7.094	0.906	RR	IC	2	LCCOMB_X19_Y16_N30	theAlu|Add0~14|datab
7.689	0.595	RF	CELL	1	LCCOMB_X19_Y16_N30	theAlu|Add0~14|cout
7.689	0.000	FF	IC	2	LCCOMB_X19_Y15_N0	theAlu|Add0~16|cin
7.769	0.080	FR	CELL	1	LCCOMB_X19_Y15_N0	theAlu|Add0~16|cout
7.769	0.000	RR	IC	2	LCCOMB_X19_Y15_N2	theAlu|Add0~18|cin
7.849	0.080	RF	CELL	1	LCCOMB_X19_Y15_N2	theAlu|Add0~18|cout
7.849	0.000	FF	IC	2	LCCOMB_X19_Y15_N4	theAlu|Add0~20|cin
7.929	0.080	FR	CELL	1	LCCOMB_X19_Y15_N4	theAlu|Add0~20|cout
7.929	0.000	RR	IC	2	LCCOMB_X19_Y15_N6	theAlu|Add0~22|cin
8.009	0.080	RF	CELL	1	LCCOMB_X19_Y15_N6	theAlu|Add0~22|cout
8.009	0.000	FF	IC	2	LCCOMB_X19_Y15_N8	theAlu|Add0~24|cin
8.089	0.080	FR	CELL	1	LCCOMB_X19_Y15_N8	theAlu|Add0~24|cout
8.089	0.000	RR	IC	2	LCCOMB_X19_Y15_N10	theAlu|Add0~26|cin
8.169	0.080	RF	CELL	1	LCCOMB_X19_Y15_N10	theAlu|Add0~26|cout
8.169	0.000	FF	IC	2	LCCOMB_X19_Y15_N12	theAlu|Add0~28|cin
8.249	0.080	FR	CELL	1	LCCOMB_X19_Y15_N12	theAlu|Add0~28|cout
8.249	0.000	RR	IC	2	LCCOMB_X19_Y15_N14	theAlu|Add0~30|cin
8.423	0.174	RF	CELL	1	LCCOMB_X19_Y15_N14	theAlu|Add0~30|cout
8.423	0.000	FF	IC	2	LCCOMB_X19_Y15_N16	theAlu|Add0~32|cin
8.503	0.080	FR	CELL	1	LCCOMB_X19_Y15_N16	theAlu|Add0~32|cout
8.503	0.000	RR	IC	2	LCCOMB_X19_Y15_N18	theAlu|Add0~34|cin
8.583	0.080	RF	CELL	1	LCCOMB_X19_Y15_N18	theAlu|Add0~34|cout
8.583	0.000	FF	IC	2	LCCOMB_X19_Y15_N20	theAlu|Add0~36|cin
8.663	0.080	FR	CELL	1	LCCOMB_X19_Y15_N20	theAlu|Add0~36|cout
8.663	0.000	RR	IC	2	LCCOMB_X19_Y15_N22	theAlu|Add0~38|cin
8.743	0.080	RF	CELL	1	LCCOMB_X19_Y15_N22	theAlu|Add0~38|cout
8.743	0.000	FF	IC	2	LCCOMB_X19_Y15_N24	theAlu|Add0~40|cin
8.823	0.080	FR	CELL	1	LCCOMB_X19_Y15_N24	theAlu|Add0~40|cout
8.823	0.000	RR	IC	2	LCCOMB_X19_Y15_N26	theAlu|Add0~42|cin
8.903	0.080	RF	CELL	1	LCCOMB_X19_Y15_N26	theAlu|Add0~42|cout

Imem 2.77 ns

Total	Incr	RF	Type	Fanout	Location	Element
18.667	15.810					data path
3.134	0.277		uTco	1	LCFF_X21_Y16_N25	inst_rom:rom|out[5]
3.134	0.000	FF	CELL	23	LCFF_X21_Y16_N25	rom|out[5]|regout
3.754	0.620	FF	IC	1	LCCOMB_X20_Y16_N20	ctrl|Mux3~1|datad
3.931	0.177	FR	CELL	29	LCCOMB_X20_Y16_N20	ctrl|Mux3~1|combout
4.877	0.946	RR	IC	1	LCCOMB_X19_Y12_N4	muxImmediateMode|Mux25~0|datad
5.055	0.178	RR	CELL	5	LCCOMB_X19_Y12_N4	muxImmediateMode|Mux25~0|combout
5.643	0.588	RR	IC	1	LCCOMB_X18_Y12_N20	theAlu|AdderInputB[6]~29|dataa
6.188	0.545	RR	CELL	1	LCCOMB_X18_Y12_N20	theAlu|AdderInputB[6]~29|combout
7.094	0.906	RR	IC	2	LCCOMB_X19_Y16_N30	theAlu|Add0~14|datab
7.689	0.595	RF	CELL	1	LCCOMB_X19_Y16_N30	theAlu|Add0~14|cout
7.689	0.000	FF	IC	2	LCCOMB_X19_Y15_N0	theAlu|Add0~16|cin
7.769	0.080	FR	CELL	1	LCCOMB_X19_Y15_N0	theAlu|Add0~16|cout
7.769	0.000	RR	IC	2	LCCOMB_X19_Y15_N2	theAlu|Add0~18|cin
7.849	0.080	RF	CELL	1	LCCOMB_X19_Y15_N2	theAlu|Add0~18|cout
7.849	0.000	FF	IC	2	LCCOMB_X19_Y15_N4	theAlu|Add0~20|cin
7.929	0.080	FR	CELL	1	LCCOMB_X19_Y15_N4	theAlu|Add0~20|cout
7.929	0.000	RR	IC	2	LCCOMB_X19_Y15_N6	theAlu|Add0~22|cin
8.009	0.080	RF	CELL	1	LCCOMB_X19_Y15_N6	theAlu|Add0~22|cout
8.009	0.000	FF	IC	2	LCCOMB_X19_Y15_N8	theAlu|Add0~24|cin
8.089	0.080	FR	CELL	1	LCCOMB_X19_Y15_N8	theAlu|Add0~24|cout
8.089	0.000	RR	IC	2	LCCOMB_X19_Y15_N10	theAlu|Add0~26|cin
8.169	0.080	RF	CELL	1	LCCOMB_X19_Y15_N10	theAlu|Add0~26|cout
8.169	0.000	FF	IC	2	LCCOMB_X19_Y15_N12	theAlu|Add0~28|cin
8.249	0.080	FR	CELL	1	LCCOMB_X19_Y15_N12	theAlu|Add0~28|cout
8.249	0.000	RR	IC	2	LCCOMB_X19_Y15_N14	theAlu|Add0~30|cin
8.423	0.174	RF	CELL	1	LCCOMB_X19_Y15_N14	theAlu|Add0~30|cout
8.423	0.000	FF	IC	2	LCCOMB_X19_Y15_N16	theAlu|Add0~32|cin
8.503	0.080	FR	CELL	1	LCCOMB_X19_Y15_N16	theAlu|Add0~32|cout
8.503	0.000	RR	IC	2	LCCOMB_X19_Y15_N18	theAlu|Add0~34|cin
8.583	0.080	RF	CELL	1	LCCOMB_X19_Y15_N18	theAlu|Add0~34|cout
8.583	0.000	FF	IC	2	LCCOMB_X19_Y15_N20	theAlu|Add0~36|cin
8.663	0.080	FR	CELL	1	LCCOMB_X19_Y15_N20	theAlu|Add0~36|cout
8.663	0.000	RR	IC	2	LCCOMB_X19_Y15_N22	theAlu|Add0~38|cin
8.743	0.080	RF	CELL	1	LCCOMB_X19_Y15_N22	theAlu|Add0~38|cout
8.743	0.000	FF	IC	2	LCCOMB_X19_Y15_N24	theAlu|Add0~40|cin
8.823	0.080	FR	CELL	1	LCCOMB_X19_Y15_N24	theAlu|Add0~40|cout
8.823	0.000	RR	IC	2	LCCOMB_X19_Y15_N26	theAlu|Add0~42|cin
8.903	0.080	RF	CELL	1	LCCOMB_X19_Y15_N26	theAlu|Add0~42|cout

Imem 2.77 ns

Ctrl 0.797 ns

Total	Incr	RF	Type	Fanout	Location	Element
18.667	15.810					data path
3.134	0.277		uTco	1	LCFF_X21_Y16_N25	inst_rom:rom|out[5]
3.134	0.000	FF	CELL	23	LCFF_X21_Y16_N25	rom|out[5]|regout
3.754	0.620	FF	IC	1	LCCOMB_X20_Y16_N20	ctrl|Mux3~1|datad
3.931	0.177	FR	CELL	29	LCCOMB_X20_Y16_N20	ctrl|Mux3~1|combout
4.877	0.946	RR	IC	1	LCCOMB_X19_Y12_N4	muxImmediateMode|Mux25~0|datad
5.055	0.178	RR	CELL	5	LCCOMB_X19_Y12_N4	muxImmediateMode|Mux25~0|combout
5.643	0.588	RR	IC	1	LCCOMB_X18_Y12_N20	theAlu|AdderInputB[6]~29|dataa
6.188	0.545	RR	CELL	1	LCCOMB_X18_Y12_N20	theAlu|AdderInputB[6]~29|combout
7.094	0.906	RR	IC	2	LCCOMB_X19_Y16_N30	theAlu|Add0~14|datab
7.689	0.595	RF	CELL	1	LCCOMB_X19_Y16_N30	theAlu|Add0~14|cout
7.689	0.000	FF	IC	2	LCCOMB_X19_Y15_N0	theAlu|Add0~16|cin
7.769	0.080	FR	CELL	1	LCCOMB_X19_Y15_N0	theAlu|Add0~16|cout
7.769	0.000	RR	IC	2	LCCOMB_X19_Y15_N2	theAlu|Add0~18|cin
7.849	0.080	RF	CELL	1	LCCOMB_X19_Y15_N2	theAlu|Add0~18|cout
7.849	0.000	FF	IC	2	LCCOMB_X19_Y15_N4	theAlu|Add0~20|cin
7.929	0.080	FR	CELL	1	LCCOMB_X19_Y15_N4	theAlu|Add0~20|cout
7.929	0.000	RR	IC	2	LCCOMB_X19_Y15_N6	theAlu|Add0~22|cin
8.009	0.080	RF	CELL	1	LCCOMB_X19_Y15_N6	theAlu|Add0~22|cout
8.009	0.000	FF	IC	2	LCCOMB_X19_Y15_N8	theAlu|Add0~24|cin
8.089	0.080	FR	CELL	1	LCCOMB_X19_Y15_N8	theAlu|Add0~24|cout
8.089	0.000	RR	IC	2	LCCOMB_X19_Y15_N10	theAlu|Add0~26|cin
8.169	0.080	RF	CELL	1	LCCOMB_X19_Y15_N10	theAlu|Add0~26|cout
8.169	0.000	FF	IC	2	LCCOMB_X19_Y15_N12	theAlu|Add0~28|cin
8.249	0.080	FR	CELL	1	LCCOMB_X19_Y15_N12	theAlu|Add0~28|cout
8.249	0.000	RR	IC	2	LCCOMB_X19_Y15_N14	theAlu|Add0~30|cin
8.423	0.174	RF	CELL	1	LCCOMB_X19_Y15_N14	theAlu|Add0~30|cout
8.423	0.000	FF	IC	2	LCCOMB_X19_Y15_N16	theAlu|Add0~32|cin
8.503	0.080	FR	CELL	1	LCCOMB_X19_Y15_N16	theAlu|Add0~32|cout
8.503	0.000	RR	IC	2	LCCOMB_X19_Y15_N18	theAlu|Add0~34|cin
8.583	0.080	RF	CELL	1	LCCOMB_X19_Y15_N18	theAlu|Add0~34|cout
8.583	0.000	FF	IC	2	LCCOMB_X19_Y15_N20	theAlu|Add0~36|cin
8.663	0.080	FR	CELL	1	LCCOMB_X19_Y15_N20	theAlu|Add0~36|cout
8.663	0.000	RR	IC	2	LCCOMB_X19_Y15_N22	theAlu|Add0~38|cin
8.743	0.080	RF	CELL	1	LCCOMB_X19_Y15_N22	theAlu|Add0~38|cout
8.743	0.000	FF	IC	2	LCCOMB_X19_Y15_N24	theAlu|Add0~40|cin
8.823	0.080	FR	CELL	1	LCCOMB_X19_Y15_N24	theAlu|Add0~40|cout
8.823	0.000	RR	IC	2	LCCOMB_X19_Y15_N26	theAlu|Add0~42|cin
8.903	0.080	RF	CELL	1	LCCOMB_X19_Y15_N26	theAlu|Add0~42|cout

Imem 2.77 ns

Ctrl 0.797 ns

ArgBMux 1.124 ns

Total	Incr	RF	Type	Fanout	Location	Element
18.667	15.810					data path
3.134	0.277		uTco	1	LCFF_X21_Y16_N25	inst_rom:rom|out[5]
3.134	0.000	FF	CELL	23	LCFF_X21_Y16_N25	rom|out[5]|regout
3.754	0.620	FF	IC	1	LCCOMB_X20_Y16_N20	ctrl|Mux3~1|datad
3.931	0.177	FR	CELL	29	LCCOMB_X20_Y16_N20	ctrl|Mux3~1|combout
4.877	0.946	RR	IC	1	LCCOMB_X19_Y12_N4	muxImmediateMode|Mux25~0|datad
5.055	0.178	RR	CELL	5	LCCOMB_X19_Y12_N4	muxImmediateMode|Mux25~0|combout
5.643	0.588	RR	IC	1	LCCOMB_X18_Y12_N20	theAlu|AdderInputB[6]~29|dataa
6.188	0.545	RR	CELL	1	LCCOMB_X18_Y12_N20	theAlu|AdderInputB[6]~29|combout
7.094	0.906	RR	IC	2	LCCOMB_X19_Y16_N30	theAlu|Add0~14|datab
7.689	0.595	RF	CELL	1	LCCOMB_X19_Y16_N30	theAlu|Add0~14|cout
7.689	0.000	FF	IC	2	LCCOMB_X19_Y15_N0	theAlu|Add0~16|cin
7.769	0.080	FR	CELL	1	LCCOMB_X19_Y15_N0	theAlu|Add0~16|cout
7.769	0.000	RR	IC	2	LCCOMB_X19_Y15_N2	theAlu|Add0~18|cin
7.849	0.080	RF	CELL	1	LCCOMB_X19_Y15_N2	theAlu|Add0~18|cout
7.849	0.000	FF	IC	2	LCCOMB_X19_Y15_N4	theAlu|Add0~20|cin
7.929	0.080	FR	CELL	1	LCCOMB_X19_Y15_N4	theAlu|Add0~20|cout
7.929	0.000	RR	IC	2	LCCOMB_X19_Y15_N6	theAlu|Add0~22|cin
8.009	0.080	RF	CELL	1	LCCOMB_X19_Y15_N6	theAlu|Add0~22|cout
8.009	0.000	FF	IC	2	LCCOMB_X19_Y15_N8	theAlu|Add0~24|cin
8.089	0.080	FR	CELL	1	LCCOMB_X19_Y15_N8	theAlu|Add0~24|cout
8.089	0.000	RR	IC	2	LCCOMB_X19_Y15_N10	theAlu|Add0~26|cin
8.169	0.080	RF	CELL	1	LCCOMB_X19_Y15_N10	theAlu|Add0~26|cout
8.169	0.000	FF	IC	2	LCCOMB_X19_Y15_N12	theAlu|Add0~28|cin
8.249	0.080	FR	CELL	1	LCCOMB_X19_Y15_N12	theAlu|Add0~28|cout
8.249	0.000	RR	IC	2	LCCOMB_X19_Y15_N14	theAlu|Add0~30|cin
8.423	0.174	RF	CELL	1	LCCOMB_X19_Y15_N14	theAlu|Add0~30|cout
8.423	0.000	FF	IC	2	LCCOMB_X19_Y15_N16	theAlu|Add0~32|cin
8.503	0.080	FR	CELL	1	LCCOMB_X19_Y15_N16	theAlu|Add0~32|cout
8.503	0.000	RR	IC	2	LCCOMB_X19_Y15_N18	theAlu|Add0~34|cin
8.583	0.080	RF	CELL	1	LCCOMB_X19_Y15_N18	theAlu|Add0~34|cout
8.583	0.000	FF	IC	2	LCCOMB_X19_Y15_N20	theAlu|Add0~36|cin
8.663	0.080	FR	CELL	1	LCCOMB_X19_Y15_N20	theAlu|Add0~36|cout
8.663	0.000	RR	IC	2	LCCOMB_X19_Y15_N22	theAlu|Add0~38|cin
8.743	0.080	RF	CELL	1	LCCOMB_X19_Y15_N22	theAlu|Add0~38|cout
8.743	0.000	FF	IC	2	LCCOMB_X19_Y15_N24	theAlu|Add0~40|cin
8.823	0.080	FR	CELL	1	LCCOMB_X19_Y15_N24	theAlu|Add0~40|cout
8.823	0.000	RR	IC	2	LCCOMB_X19_Y15_N26	theAlu|Add0~42|cin
8.903	0.080	RF	CELL	1	LCCOMB_X19_Y15_N26	theAlu|Add0~42|cout

Imem 2.77 ns

Ctrl 0.797 ns

ArgBMux 1.124 ns

ALU 6.527ns

8.983	0.080	FR	CELL	1	LCCOMB_X19_Y15_N28	theAlu|Add0~44|cout
8.983	0.000	RR	IC	2	LCCOMB_X19_Y15_N30	theAlu|Add0~46|cin
9.144	0.161	RF	CELL	1	LCCOMB_X19_Y15_N30	theAlu|Add0~46|cout
9.144	0.000	FF	IC	2	LCCOMB_X19_Y14_N0	theAlu|Add0~48|cin
9.224	0.080	FR	CELL	1	LCCOMB_X19_Y14_N0	theAlu|Add0~48|cout
9.224	0.000	RR	IC	2	LCCOMB_X19_Y14_N2	theAlu|Add0~50|cin
9.304	0.080	RF	CELL	1	LCCOMB_X19_Y14_N2	theAlu|Add0~50|cout
9.304	0.000	FF	IC	2	LCCOMB_X19_Y14_N4	theAlu|Add0~52|cin
9.384	0.080	FR	CELL	1	LCCOMB_X19_Y14_N4	theAlu|Add0~52|cout
9.384	0.000	RR	IC	2	LCCOMB_X19_Y14_N6	theAlu|Add0~54|cin
9.464	0.080	RF	CELL	1	LCCOMB_X19_Y14_N6	theAlu|Add0~54|cout
9.464	0.000	FF	IC	2	LCCOMB_X19_Y14_N8	theAlu|Add0~56|cin
9.544	0.080	FR	CELL	1	LCCOMB_X19_Y14_N8	theAlu|Add0~56|cout
9.544	0.000	RR	IC	2	LCCOMB_X19_Y14_N10	theAlu|Add0~58|cin
9.624	0.080	RF	CELL	1	LCCOMB_X19_Y14_N10	theAlu|Add0~58|cout
9.624	0.000	FF	IC	2	LCCOMB_X19_Y14_N12	theAlu|Add0~60|cin
9.704	0.080	FR	CELL	1	LCCOMB_X19_Y14_N12	theAlu|Add0~60|cout
9.704	0.000	RR	IC	2	LCCOMB_X19_Y14_N14	theAlu|Add0~62|cin
9.878	0.174	RF	CELL	1	LCCOMB_X19_Y14_N14	theAlu|Add0~62|cout
9.878	0.000	FF	IC	1	LCCOMB_X19_Y14_N16	theAlu|Add0~64|cin
10.336	0.458	FF	CELL	1	LCCOMB_X19_Y14_N16	theAlu|Add0~64|combout
11.260	0.924	FF	IC	1	LCCOMB_X19_Y18_N0	theAlu|O_out[31]~39|datac
11.582	0.322	FF	CELL	12	LCCOMB_X19_Y18_N0	theAlu|O_out[31]~39|combout
12.517	0.935	FF	IC	1	LCCOMB_X20_Y15_N4	dataMem|Equal0~0|datad
12.694	0.177	FR	CELL	1	LCCOMB_X20_Y15_N4	dataMem|Equal0~0|combout
13.004	0.310	RR	IC	1	LCCOMB_X20_Y15_N12	dataMem|Equal0~4|datac
13.326	0.322	RR	CELL	31	LCCOMB_X20_Y15_N12	dataMem|Equal0~4|combout
14.615	1.289	RR	IC	1	LCCOMB_X18_Y13_N4	muxWriteRegData|O_out[3]~3|datad
14.793	0.178	RR	CELL	7	LCCOMB_X18_Y13_N4	muxWriteRegData|O_out[3]~3|combout
16.071	1.278	RR	IC	1	LCCOMB_X19_Y18_N24	muxWriteRegData|O_out[2]|datac
16.393	0.322	RR	CELL	1	LCCOMB_X19_Y18_N24	muxWriteRegData|O_out[2]|combout
17.243	0.850	RR	IC	1	LCCOMB_X18_Y16_N0	rf|regs~8|datad
17.421	0.178	RR	CELL	3	LCCOMB_X18_Y16_N0	rf|regs~8|combout
18.254	0.833	RR	IC	1	LCFF_X20_Y16_N27	rf|regs[0][2]|sdata
18.667	0.413	RR	CELL	1	LCFF_X20_Y16_N27	register_file:rf|regs[0][2]

Dmem 1.744 ns

8.983	0.080	FR	CELL	1	LCCOMB_X19_Y15_N28	theAlu|Add0~44|cout
8.983	0.000	RR	IC	2	LCCOMB_X19_Y15_N30	theAlu|Add0~46|cin
9.144	0.161	RF	CELL	1	LCCOMB_X19_Y15_N30	theAlu|Add0~46|cout
9.144	0.000	FF	IC	2	LCCOMB_X19_Y14_N0	theAlu|Add0~48|cin
9.224	0.080	FR	CELL	1	LCCOMB_X19_Y14_N0	theAlu|Add0~48|cout
9.224	0.000	RR	IC	2	LCCOMB_X19_Y14_N2	theAlu|Add0~50|cin
9.304	0.080	RF	CELL	1	LCCOMB_X19_Y14_N2	theAlu|Add0~50|cout
9.304	0.000	FF	IC	2	LCCOMB_X19_Y14_N4	theAlu|Add0~52|cin
9.384	0.080	FR	CELL	1	LCCOMB_X19_Y14_N4	theAlu|Add0~52|cout
9.384	0.000	RR	IC	2	LCCOMB_X19_Y14_N6	theAlu|Add0~54|cin
9.464	0.080	RF	CELL	1	LCCOMB_X19_Y14_N6	theAlu|Add0~54|cout
9.464	0.000	FF	IC	2	LCCOMB_X19_Y14_N8	theAlu|Add0~56|cin
9.544	0.080	FR	CELL	1	LCCOMB_X19_Y14_N8	theAlu|Add0~56|cout
9.544	0.000	RR	IC	2	LCCOMB_X19_Y14_N10	theAlu|Add0~58|cin
9.624	0.080	RF	CELL	1	LCCOMB_X19_Y14_N10	theAlu|Add0~58|cout
9.624	0.000	FF	IC	2	LCCOMB_X19_Y14_N12	theAlu|Add0~60|cin
9.704	0.080	FR	CELL	1	LCCOMB_X19_Y14_N12	theAlu|Add0~60|cout
9.704	0.000	RR	IC	2	LCCOMB_X19_Y14_N14	theAlu|Add0~62|cin
9.878	0.174	RF	CELL	1	LCCOMB_X19_Y14_N14	theAlu|Add0~62|cout
9.878	0.000	FF	IC	1	LCCOMB_X19_Y14_N16	theAlu|Add0~64|cin
10.336	0.458	FF	CELL	1	LCCOMB_X19_Y14_N16	theAlu|Add0~64|combout
11.260	0.924	FF	IC	1	LCCOMB_X19_Y18_N0	theAlu|O_out[31]~39|datac
11.582	0.322	FF	CELL	12	LCCOMB_X19_Y18_N0	theAlu|O_out[31]~39|combout
12.517	0.935	FF	IC	1	LCCOMB_X20_Y15_N4	dataMem|Equal0~0|datad
12.694	0.177	FR	CELL	1	LCCOMB_X20_Y15_N4	dataMem|Equal0~0|combout
13.004	0.310	RR	IC	1	LCCOMB_X20_Y15_N12	dataMem|Equal0~4|datac
13.326	0.322	RR	CELL	31	LCCOMB_X20_Y15_N12	dataMem|Equal0~4|combout
14.615	1.289	RR	IC	1	LCCOMB_X18_Y13_N4	muxWriteRegData|O_out[3]~3|datad
14.793	0.178	RR	CELL	7	LCCOMB_X18_Y13_N4	muxWriteRegData|O_out[3]~3|combout
16.071	1.278	RR	IC	1	LCCOMB_X19_Y18_N24	muxWriteRegData|O_out[2]|datac
16.393	0.322	RR	CELL	1	LCCOMB_X19_Y18_N24	muxWriteRegData|O_out[2]|combout
17.243	0.850	RR	IC	1	LCCOMB_X18_Y16_N0	rf|regs~8|datad
17.421	0.178	RR	CELL	3	LCCOMB_X18_Y16_N0	rf|regs~8|combout
18.254	0.833	RR	IC	1	LCFF_X20_Y16_N27	rf|regs[0][2]|sdata
18.667	0.413	RR	CELL	1	LCFF_X20_Y16_N27	register_file:rf|regs[0][2]

8.983	0.080	FR	CELL	1		LCCOMB_X19_Y15_N28	theAlu|Add0~44|cout
8.983	0.000	RR	IC	2		LCCOMB_X19_Y15_N30	theAlu|Add0~46|cin
9.144	0.161	RF	CELL	1		LCCOMB_X19_Y15_N30	theAlu|Add0~46|cout
9.144	0.000	FF	IC	2		LCCOMB_X19_Y14_N0	theAlu|Add0~48|cin
9.224	0.080	FR	CELL	1		LCCOMB_X19_Y14_N0	theAlu|Add0~48|cout
9.224	0.000	RR	IC	2		LCCOMB_X19_Y14_N2	theAlu|Add0~50|cin
9.304	0.080	RF	CELL	1		LCCOMB_X19_Y14_N2	theAlu|Add0~50|cout
9.304	0.000	FF	IC	2		LCCOMB_X19_Y14_N4	theAlu|Add0~52|cin
9.384	0.080	FR	CELL	1		LCCOMB_X19_Y14_N4	theAlu|Add0~52|cout
9.384	0.000	RR	IC	2		LCCOMB_X19_Y14_N6	theAlu|Add0~54|cin
9.464	0.080	RF	CELL	1		LCCOMB_X19_Y14_N6	theAlu|Add0~54|cout
9.464	0.000	FF	IC	2		LCCOMB_X19_Y14_N8	theAlu|Add0~56|cin
9.544	0.080	FR	CELL	1		LCCOMB_X19_Y14_N8	theAlu|Add0~56|cout
9.544	0.000	RR	IC	2		LCCOMB_X19_Y14_N10	theAlu|Add0~58|cin
9.624	0.080	RF	CELL	1		LCCOMB_X19_Y14_N10	theAlu|Add0~58|cout
9.624	0.000	FF	IC	2		LCCOMB_X19_Y14_N12	theAlu|Add0~60|cin
9.704	0.080	FR	CELL	1		LCCOMB_X19_Y14_N12	theAlu|Add0~60|cout
9.704	0.000	RR	IC	2		LCCOMB_X19_Y14_N14	theAlu|Add0~62|cin
9.878	0.174	RF	CELL	1		LCCOMB_X19_Y14_N14	theAlu|Add0~62|cout
9.878	0.000	FF	IC	1		LCCOMB_X19_Y14_N16	theAlu|Add0~64|cin
10.336	0.458	FF	CELL	1		LCCOMB_X19_Y14_N16	theAlu|Add0~64|combout
11.260	0.924	FF	IC	1		LCCOMB_X19_Y18_N0	theAlu|O_out[31]~39|datac
11.582	0.322	FF	CELL	12		LCCOMB_X19_Y18_N0	theAlu|O_out[31]~39|combout
12.517	0.935	FF	IC	1		LCCOMB_X20_Y15_N4	dataMem|Equal0~0|datad
12.694	0.177	FR	CELL	1		LCCOMB_X20_Y15_N4	dataMem|Equal0~0|combout
13.004	0.310	RR	IC	1		LCCOMB_X20_Y15_N12	dataMem|Equal0~4|datac
13.326	0.322	RR	CELL	31		LCCOMB_X20_Y15_N12	dataMem|Equal0~4|combout
14.615	1.289	RR	IC	1		LCCOMB_X18_Y13_N4	muxWriteRegData|O_out[3]~3|datad
14.793	0.178	RR	CELL	7		LCCOMB_X18_Y13_N4	muxWriteRegData|O_out[3]~3|combout
16.071	1.278	RR	IC	1		LCCOMB_X19_Y18_N24	muxWriteRegData|O_out[2]|datac
16.393	0.322	RR	CELL	1		LCCOMB_X19_Y18_N24	muxWriteRegData|O_out[2]|combout
17.243	0.850	RR	IC	1		LCCOMB_X18_Y16_N0	rf|regs~8|datad
17.421	0.178	RR	CELL	3		LCCOMB_X18_Y16_N0	rf|regs~8|combout
18.254	0.833	RR	IC	1		LCFF_X20_Y16_N27	rf|regs[0][2]|sdata
18.667	0.413	RR	CELL	1		LCFF_X20_Y16_N27	register_file:rf|regs[0][2]

ALU 6.527 ns

8.983	0.080	FR	CELL	1	LCCOMB_X19_Y15_N28	theAlu|Add0~44|cout
8.983	0.000	RR	IC	2	LCCOMB_X19_Y15_N30	theAlu|Add0~46|cin
9.144	0.161	RF	CELL	1	LCCOMB_X19_Y15_N30	theAlu|Add0~46|cout
9.144	0.000	FF	IC	2	LCCOMB_X19_Y14_N0	theAlu|Add0~48|cin
9.224	0.080	FR	CELL	1	LCCOMB_X19_Y14_N0	theAlu|Add0~48|cout
9.224	0.000	RR	IC	2	LCCOMB_X19_Y14_N2	theAlu|Add0~50|cin
9.304	0.080	RF	CELL	1	LCCOMB_X19_Y14_N2	theAlu|Add0~50|cout
9.304	0.000	FF	IC	2	LCCOMB_X19_Y14_N4	theAlu|Add0~52|cin
9.384	0.080	FR	CELL	1	LCCOMB_X19_Y14_N4	theAlu|Add0~52|cout
9.384	0.000	RR	IC	2	LCCOMB_X19_Y14_N6	theAlu|Add0~54|cin
9.464	0.080	RF	CELL	1	LCCOMB_X19_Y14_N6	theAlu|Add0~54|cout
9.464	0.000	FF	IC	2	LCCOMB_X19_Y14_N8	theAlu|Add0~56|cin
9.544	0.080	FR	CELL	1	LCCOMB_X19_Y14_N8	theAlu|Add0~56|cout
9.544	0.000	RR	IC	2	LCCOMB_X19_Y14_N10	theAlu|Add0~58|cin
9.624	0.080	RF	CELL	1	LCCOMB_X19_Y14_N10	theAlu|Add0~58|cout
9.624	0.000	FF	IC	2	LCCOMB_X19_Y14_N12	theAlu|Add0~60|cin
9.704	0.080	FR	CELL	1	LCCOMB_X19_Y14_N12	theAlu|Add0~60|cout
9.704	0.000	RR	IC	2	LCCOMB_X19_Y14_N14	theAlu|Add0~62|cin
9.878	0.174	RF	CELL	1	LCCOMB_X19_Y14_N14	theAlu|Add0~62|cout
9.878	0.000	FF	IC	1	LCCOMB_X19_Y14_N16	theAlu|Add0~64|cin
10.336	0.458	FF	CELL	1	LCCOMB_X19_Y14_N16	theAlu|Add0~64|combout
11.260	0.924	FF	IC	1	LCCOMB_X19_Y18_N0	theAlu|O_out[31]~39|datac
11.582	0.322	FF	CELL	12	LCCOMB_X19_Y18_N0	theAlu|O_out[31]~39|combout
12.517	0.935	FF	IC	1	LCCOMB_X20_Y15_N4	dataMem|Equal0~0|datad
12.694	0.177	FR	CELL	1	LCCOMB_X20_Y15_N4	dataMem|Equal0~0|combout
13.004	0.310	RR	IC	1	LCCOMB_X20_Y15_N12	dataMem|Equal0~4|datac
13.326	0.322	RR	CELL	31	LCCOMB_X20_Y15_N12	dataMem|Equal0~4|combout
14.615	1.289	RR	IC	1	LCCOMB_X18_Y13_N4	muxWriteRegData|O_out[3]~3|datad
14.793	0.178	RR	CELL	7	LCCOMB_X18_Y13_N4	muxWriteRegData|O_out[3]~3|combout
16.071	1.278	RR	IC	1	LCCOMB_X19_Y18_N24	muxWriteRegData|O_out[2]|datac
16.393	0.322	RR	CELL	1	LCCOMB_X19_Y18_N24	muxWriteRegData|O_out[2]|combout
17.243	0.850	RR	IC	1	LCCOMB_X18_Y16_N0	rf|regs~8|datad
17.421	0.178	RR	CELL	3	LCCOMB_X18_Y16_N0	rf|regs~8|combout
18.254	0.833	RR	IC	1	LCFF_X20_Y16_N27	rf|regs[0][2]|sdata
18.667	0.413	RR	CELL	1	LCFF_X20_Y16_N27	register_file:rf|regs[0][2]

8.983	0.080	FR	CELL	1	LCCOMB_X19_Y15_N28	theAlu|Add0~44|cout
8.983	0.000	RR	IC	2	LCCOMB_X19_Y15_N30	theAlu|Add0~46|cin
9.144	0.161	RF	CELL	1	LCCOMB_X19_Y15_N30	theAlu|Add0~46|cout
9.144	0.000	FF	IC	2	LCCOMB_X19_Y14_N0	theAlu|Add0~48|cin
9.224	0.080	FR	CELL	1	LCCOMB_X19_Y14_N0	theAlu|Add0~48|cout
9.224	0.000	RR	IC	2	LCCOMB_X19_Y14_N2	theAlu|Add0~50|cin
9.304	0.080	RF	CELL	1	LCCOMB_X19_Y14_N2	theAlu|Add0~50|cout
9.304	0.000	FF	IC	2	LCCOMB_X19_Y14_N4	theAlu|Add0~52|cin
9.384	0.080	FR	CELL	1	LCCOMB_X19_Y14_N4	theAlu|Add0~52|cout
9.384	0.000	RR	IC	2	LCCOMB_X19_Y14_N6	theAlu|Add0~54|cin
9.464	0.080	RF	CELL	1	LCCOMB_X19_Y14_N6	theAlu|Add0~54|cout
9.464	0.000	FF	IC	2	LCCOMB_X19_Y14_N8	theAlu|Add0~56|cin
9.544	0.080	FR	CELL	1	LCCOMB_X19_Y14_N8	theAlu|Add0~56|cout
9.544	0.000	RR	IC	2	LCCOMB_X19_Y14_N10	theAlu|Add0~58|cin
9.624	0.080	RF	CELL	1	LCCOMB_X19_Y14_N10	theAlu|Add0~58|cout
9.624	0.000	FF	IC	2	LCCOMB_X19_Y14_N12	theAlu|Add0~60|cin
9.704	0.080	FR	CELL	1	LCCOMB_X19_Y14_N12	theAlu|Add0~60|cout
9.704	0.000	RR	IC	2	LCCOMB_X19_Y14_N14	theAlu|Add0~62|cin
9.878	0.174	RF	CELL	1	LCCOMB_X19_Y14_N14	theAlu|Add0~62|cout
9.878	0.000	FF	IC	1	LCCOMB_X19_Y14_N16	theAlu|Add0~64|cin
10.336	0.458	FF	CELL	1	LCCOMB_X19_Y14_N16	theAlu|Add0~64|combout
11.260	0.924	FF	IC	1	LCCOMB_X19_Y18_N0	theAlu|O_out[31]~39|datac
11.582	0.322	FF	CELL	12	LCCOMB_X19_Y18_N0	theAlu|O_out[31]~39|combout
12.517	0.935	FF	IC	1	LCCOMB_X20_Y15_N4	dataMem|Equal0~0|datad
12.694	0.177	FR	CELL	1	LCCOMB_X20_Y15_N4	dataMem|Equal0~0|combout
13.004	0.310	RR	IC	1	LCCOMB_X20_Y15_N12	dataMem|Equal0~4|datac
13.326	0.322	RR	CELL	31	LCCOMB_X20_Y15_N12	dataMem|Equal0~4|combout
14.615	1.289	RR	IC	1	LCCOMB_X18_Y13_N4	muxWriteRegData|O_out[3]~3|datad
14.793	0.178	RR	CELL	7	LCCOMB_X18_Y13_N4	muxWriteRegData|O_out[3]~3|combout
16.071	1.278	RR	IC	1	LCCOMB_X19_Y18_N24	muxWriteRegData|O_out[2]|datac
16.393	0.322	RR	CELL	1	LCCOMB_X19_Y18_N24	muxWriteRegData|O_out[2]|combout
17.243	0.850	RR	IC	1	LCCOMB_X18_Y16_N0	rf|regs~8|datad
17.421	0.178	RR	CELL	3	LCCOMB_X18_Y16_N0	rf|regs~8|combout
18.254	0.833	RR	IC	1	LCFF_X20_Y16_N27	rf|regs[0][2]|sdata
18.667	0.413	RR	CELL	1	LCFF_X20_Y16_N27	register_file:rf|regs[0][2]

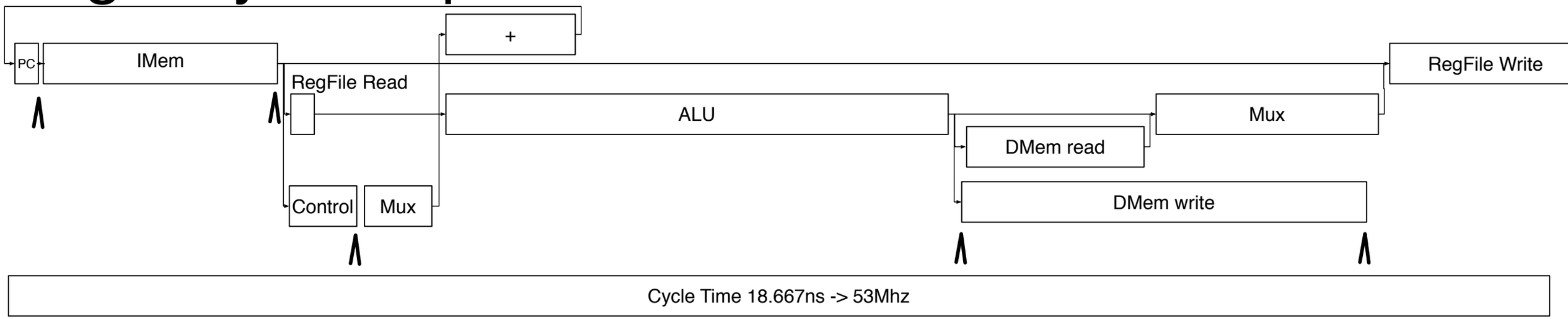
WriteRegMux
3.067 ns

8.983	0.080	FR	CELL	1	LCCOMB_X19_Y15_N28	theAlu|Add0~44|cout
8.983	0.000	RR	IC	2	LCCOMB_X19_Y15_N30	theAlu|Add0~46|cin
9.144	0.161	RF	CELL	1	LCCOMB_X19_Y15_N30	theAlu|Add0~46|cout
9.144	0.000	FF	IC	2	LCCOMB_X19_Y14_N0	theAlu|Add0~48|cin
9.224	0.080	FR	CELL	1	LCCOMB_X19_Y14_N0	theAlu|Add0~48|cout
9.224	0.000	RR	IC	2	LCCOMB_X19_Y14_N2	theAlu|Add0~50|cin
9.304	0.080	RF	CELL	1	LCCOMB_X19_Y14_N2	theAlu|Add0~50|cout
9.304	0.000	FF	IC	2	LCCOMB_X19_Y14_N4	theAlu|Add0~52|cin
9.384	0.080	FR	CELL	1	LCCOMB_X19_Y14_N4	theAlu|Add0~52|cout
9.384	0.000	RR	IC	2	LCCOMB_X19_Y14_N6	theAlu|Add0~54|cin
9.464	0.080	RF	CELL	1	LCCOMB_X19_Y14_N6	theAlu|Add0~54|cout
9.464	0.000	FF	IC	2	LCCOMB_X19_Y14_N8	theAlu|Add0~56|cin
9.544	0.080	FR	CELL	1	LCCOMB_X19_Y14_N8	theAlu|Add0~56|cout
9.544	0.000	RR	IC	2	LCCOMB_X19_Y14_N10	theAlu|Add0~58|cin
9.624	0.080	RF	CELL	1	LCCOMB_X19_Y14_N10	theAlu|Add0~58|cout
9.624	0.000	FF	IC	2	LCCOMB_X19_Y14_N12	theAlu|Add0~60|cin
9.704	0.080	FR	CELL	1	LCCOMB_X19_Y14_N12	theAlu|Add0~60|cout
9.704	0.000	RR	IC	2	LCCOMB_X19_Y14_N14	theAlu|Add0~62|cin
9.878	0.174	RF	CELL	1	LCCOMB_X19_Y14_N14	theAlu|Add0~62|cout
9.878	0.000	FF	IC	1	LCCOMB_X19_Y14_N16	theAlu|Add0~64|cin
10.336	0.458	FF	CELL	1	LCCOMB_X19_Y14_N16	theAlu|Add0~64|combout
11.260	0.924	FF	IC	1	LCCOMB_X19_Y18_N0	theAlu|O_out[31]~39|datac
11.582	0.322	FF	CELL	12	LCCOMB_X19_Y18_N0	theAlu|O_out[31]~39|combout
12.517	0.935	FF	IC	1	LCCOMB_X20_Y15_N4	dataMem|Equal0~0|datad
12.694	0.177	FR	CELL	1	LCCOMB_X20_Y15_N4	dataMem|Equal0~0|combout
13.004	0.310	RR	IC	1	LCCOMB_X20_Y15_N12	dataMem|Equal0~4|datac
13.326	0.322	RR	CELL	31	LCCOMB_X20_Y15_N12	dataMem|Equal0~4|combout
14.615	1.289	RR	IC	1	LCCOMB_X18_Y13_N4	muxWriteRegData|O_out[3]~3|datad
14.793	0.178	RR	CELL	7	LCCOMB_X18_Y13_N4	muxWriteRegData|O_out[3]~3|combout
16.071	1.278	RR	IC	1	LCCOMB_X19_Y18_N24	muxWriteRegData|O_out[2]|datac
16.393	0.322	RR	CELL	1	LCCOMB_X19_Y18_N24	muxWriteRegData|O_out[2]|combout
17.243	0.850	RR	IC	1	LCCOMB_X18_Y16_N0	rf|regs~8|datad
17.421	0.178	RR	CELL	3	LCCOMB_X18_Y16_N0	rf|regs~8|combout
18.254	0.833	RR	IC	1	LCFF_X20_Y16_N27	rf|regs[0][2]|sdata
18.667	0.413	RR	CELL	1	LCFF_X20_Y16_N27	register_file:rf|regs[0][2]

8.983	0.080	FR	CELL	1	LCCOMB_X19_Y15_N28	theAlu|Add0~44|cout
8.983	0.000	RR	IC	2	LCCOMB_X19_Y15_N30	theAlu|Add0~46|cin
9.144	0.161	RF	CELL	1	LCCOMB_X19_Y15_N30	theAlu|Add0~46|cout
9.144	0.000	FF	IC	2	LCCOMB_X19_Y14_N0	theAlu|Add0~48|cin
9.224	0.080	FR	CELL	1	LCCOMB_X19_Y14_N0	theAlu|Add0~48|cout
9.224	0.000	RR	IC	2	LCCOMB_X19_Y14_N2	theAlu|Add0~50|cin
9.304	0.080	RF	CELL	1	LCCOMB_X19_Y14_N2	theAlu|Add0~50|cout
9.304	0.000	FF	IC	2	LCCOMB_X19_Y14_N4	theAlu|Add0~52|cin
9.384	0.080	FR	CELL	1	LCCOMB_X19_Y14_N4	theAlu|Add0~52|cout
9.384	0.000	RR	IC	2	LCCOMB_X19_Y14_N6	theAlu|Add0~54|cin
9.464	0.080	RF	CELL	1	LCCOMB_X19_Y14_N6	theAlu|Add0~54|cout
9.464	0.000	FF	IC	2	LCCOMB_X19_Y14_N8	theAlu|Add0~56|cin
9.544	0.080	FR	CELL	1	LCCOMB_X19_Y14_N8	theAlu|Add0~56|cout
9.544	0.000	RR	IC	2	LCCOMB_X19_Y14_N10	theAlu|Add0~58|cin
9.624	0.080	RF	CELL	1	LCCOMB_X19_Y14_N10	theAlu|Add0~58|cout
9.624	0.000	FF	IC	2	LCCOMB_X19_Y14_N12	theAlu|Add0~60|cin
9.704	0.080	FR	CELL	1	LCCOMB_X19_Y14_N12	theAlu|Add0~60|cout
9.704	0.000	RR	IC	2	LCCOMB_X19_Y14_N14	theAlu|Add0~62|cin
9.878	0.174	RF	CELL	1	LCCOMB_X19_Y14_N14	theAlu|Add0~62|cout
9.878	0.000	FF	IC	1	LCCOMB_X19_Y14_N16	theAlu|Add0~64|cin
10.336	0.458	FF	CELL	1	LCCOMB_X19_Y14_N16	theAlu|Add0~64|combout
11.260	0.924	FF	IC	1	LCCOMB_X19_Y18_N0	theAlu|O_out[31]~39|datac
11.582	0.322	FF	CELL	12	LCCOMB_X19_Y18_N0	theAlu|O_out[31]~39|combout
12.517	0.935	FF	IC	1	LCCOMB_X20_Y15_N4	dataMem|Equal0~0|datad
12.694	0.177	FR	CELL	1	LCCOMB_X20_Y15_N4	dataMem|Equal0~0|combout
13.004	0.310	RR	IC	1	LCCOMB_X20_Y15_N12	dataMem|Equal0~4|datac
13.326	0.322	RR	CELL	31	LCCOMB_X20_Y15_N12	dataMem|Equal0~4|combout
14.615	1.289	RR	IC	1	LCCOMB_X18_Y13_N4	muxWriteRegData|O_out[3]~3|datad
14.793	0.178	RR	CELL	7	LCCOMB_X18_Y13_N4	muxWriteRegData|O_out[3]~3|combout
16.071	1.278	RR	IC	1	LCCOMB_X19_Y18_N24	muxWriteRegData|O_out[2]|datac
16.393	0.322	RR	CELL	1	LCCOMB_X19_Y18_N24	muxWriteRegData|O_out[2]|combout
17.243	0.850	RR	IC	1	LCCOMB_X18_Y16_N0	rf|regs~8|datad
17.421	0.178	RR	CELL	3	LCCOMB_X18_Y16_N0	rf|regs~8|combout
18.254	0.833	RR	IC	1	LCFF_X20_Y16_N27	rf|regs[0][2]|sdata
18.667	0.413	RR	CELL	1	LCFF_X20_Y16_N27	register_file:rf|regs[0][2]

RegFile 2.27 ns

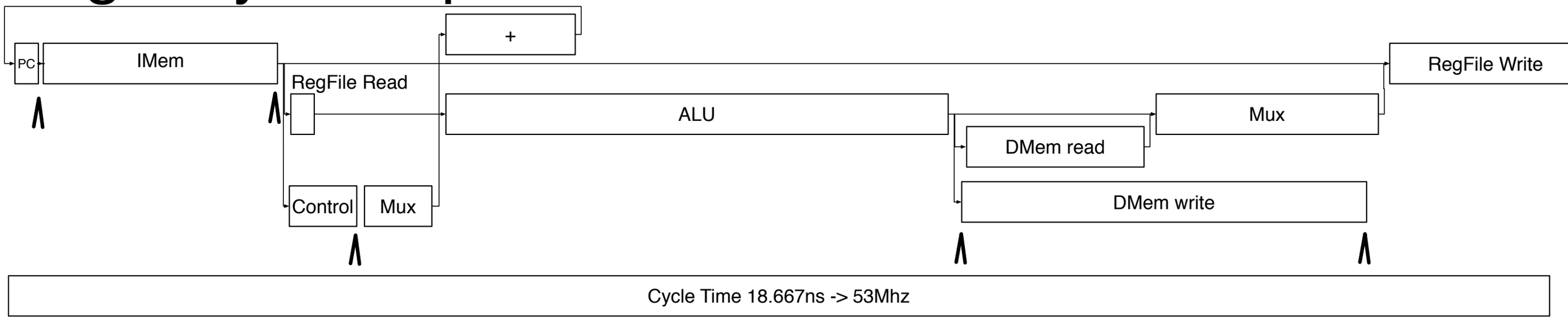
Single-cycle Implementation to scale



Ideal 5-stage Pipeline (3.733ns -> 267Mhz)

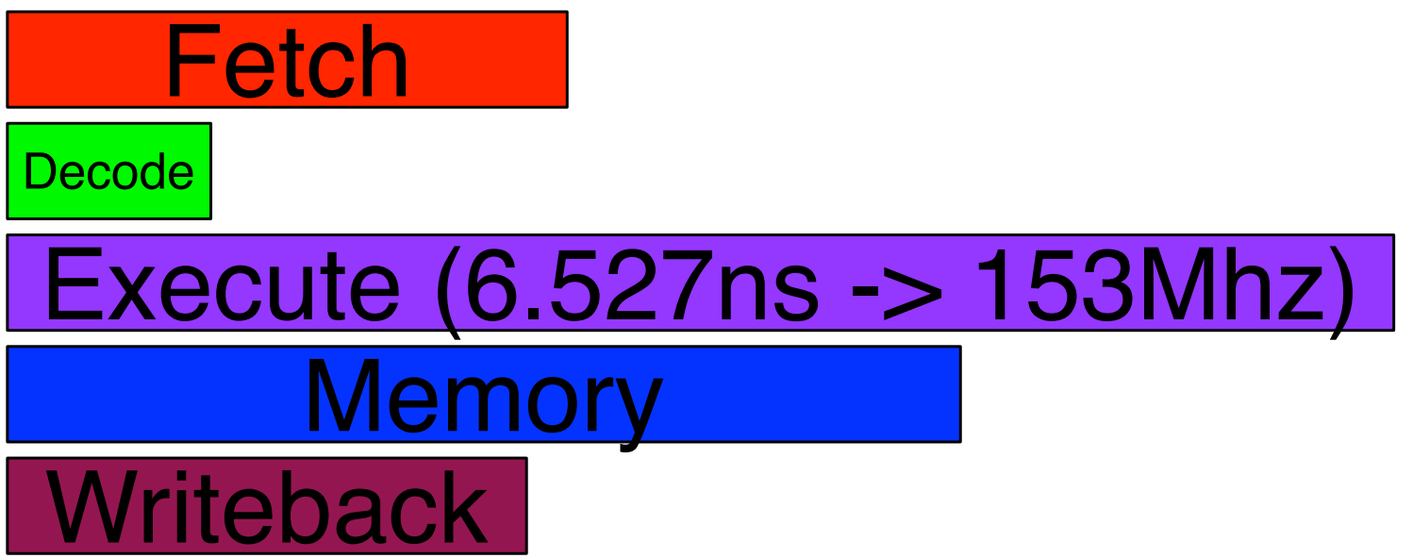
- 18.667ns -> 3.733ns == 80% reduction in CT
- Lold = IC * CPI * CTold
- Lnew = IC * CPI * CTnew
- CTnew = 0.2 * CTold
- Lnew = 0.2 * Lold
- Speed up = Lold/Lnew = 5x

Single-cycle Implementation to scale



Ideal 5-stage Pipeline (3.733ns -> 267Mhz)

Realistic 5-stage Pipeline



- Pipe stages are imbalanced
- Longest == 6.5ns
- Shortest == 0.79ns
- Speedup = 18.667/6.5 == 2.8x

Quiz 5

Question 1: True/False

Average Score 1.73134 points

The ideal CPI for a pipelined CPU is larger than for a non-pipelined CPU.

Correct	Answers	Percent Answered
	True	13.433%
☑	False	86.567%
	Unanswered	0%

Question 2: Multiple Answer

Average Score 2.59701 points

Which of the following can lead to imperfect pipelining?

Correct	Answers	Percent Correct	Percent Incorrect
	Instruction mix.	53.731%	46.269%
☑	Pipeline register setup time.	83.582%	16.418%
	Compiler optimizations.	92.537%	7.463%
☑	Complex, long-latency datapath components.	89.552%	10.448%

Question 3: Multiple Answer

Average Score 3.53731 points

Which of the following is true of VLIW processors?

Correct	Answers	Percent Correct	Percent Incorrect
☑	Each instruction word contains multiple instructions.	92.537%	7.463%
☑	The instructions in an instruction word execute simultaneously.	92.537%	7.463%
☑	VLIW machines are challenging for compilers to utilize effectively.	92.537%	7.463%
☑	VLIW machines have been successful in some commercial domains.	76.119%	23.881%

Question 4: Multiple Choice

Average Score 3.46269 points

Which of the following is true about CISC instruction sets?

Correct		Percent Answered
	It is prohibitively expensive to build fast CISC processors.	0%
	CISC instructions sets are more complex than RISC instruction sets.	2.985%
	RISC instruction sets are more complex than CISC instruction sets.	5.97%
	Based on their use in today's devices, it appears that RISC instruction sets are better suited to mobile computing than CISC ISAs.	1.493%
	Both 1 and 4.	2.985%
☑	Both 2 and 4.	86.567%
	Unanswered	0%

Pipelining is Tricky

- Simple pipelining is easy
  - If the data flows in one direction only
  - If the stages are independent
  - In fact the tool can do this automatically via "retiming" (If you are curious, experiment with this in Quartus).
- Not so, for processors.
  - Branch instructions affect the next PC -- backward flow
  - Instructions need values computed by previous instructions -- not independent

Not just tricky, Hazardous!

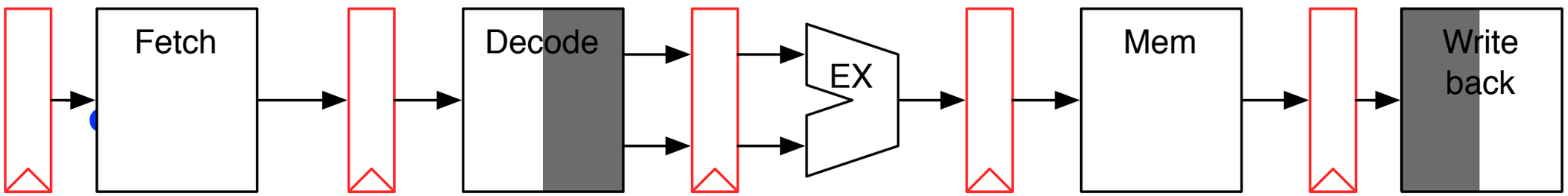
- Hazards are situations where pipelining does not work as elegantly as we would like
- Three kinds
  - Structural hazards -- we have run out of a hardware resource.
  - Data hazards -- an input is not available on the cycle it is needed.
  - Control hazards -- the next instruction is not known.
- Dealing with hazards increases complexity or decreases performance (or both)
- Dealing efficienctly with hazards is much of what makes processor design hard.
  - That, and the Quartus tools ;-)

Hazards: Key Points

- Hazards cause imperfect pipelining
  - They prevent us from achieving CPI = 1
  - They are generally causes by “counter flow” data dependences in the pipeline
- Three kinds
  - Structural -- contention for hardware resources
  - Data -- a data value is not available when/where it is needed.
  - Control -- the next instruction to execute is not known.
- Two ways to deal with hazards
  - Removal -- add hardware and/or complexity to work around the hazard so it does not occur
  - Stall -- Hold up the execution of new instructions. Let the older instructions finish, so the hazard will clear.

A Structural Hazard

- Both the decode and write back stage have to access the register file.
- There is only one registers file. A structural hazard!!
- Solution: Write early, read late
  - Writes occur at the clock edge and complete long before the end of the cycle
  - This leave enough time for the outputs to settle for the reads.
- Hazard avoided!



Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
  - Register values
  - Also memory accesses (more on this later)

```
add $s0, $t0, $t1

sub $t2, $s0, $t3

add $t3, $s0, $t4

add $t3, $t2, $t4
```

Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
  - Register values
  - Also memory accesses (more on this later)

```
add $s0, $t0, $t1

sub $t2, $s0, $t3

add $t3, $s0, $t4

add $t3, $t2, $t4
```

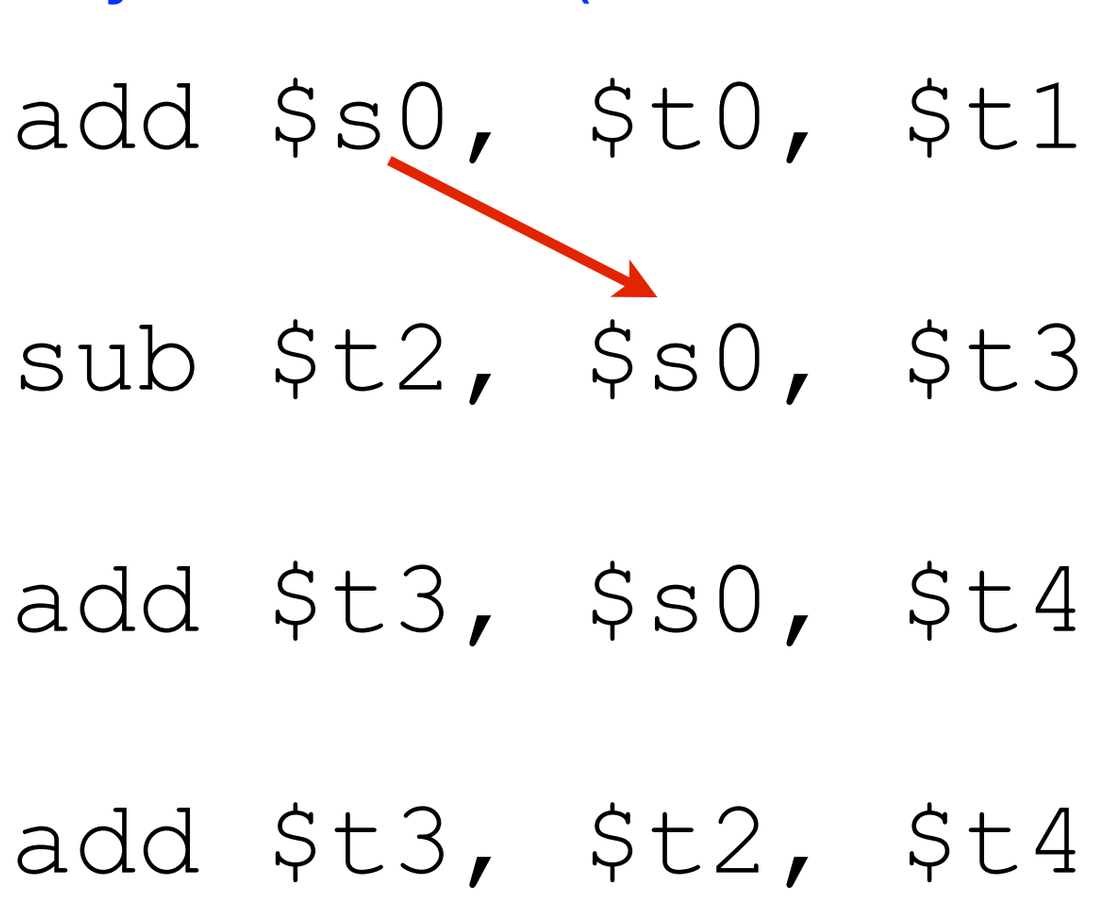
Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
  - Register values
  - Also memory accesses (more on this later)

```
add $s0, $t0, $t1

sub $t2, $s0, $t3

add $t3, $s0, $t4

add $t3, $t2, $t4
```

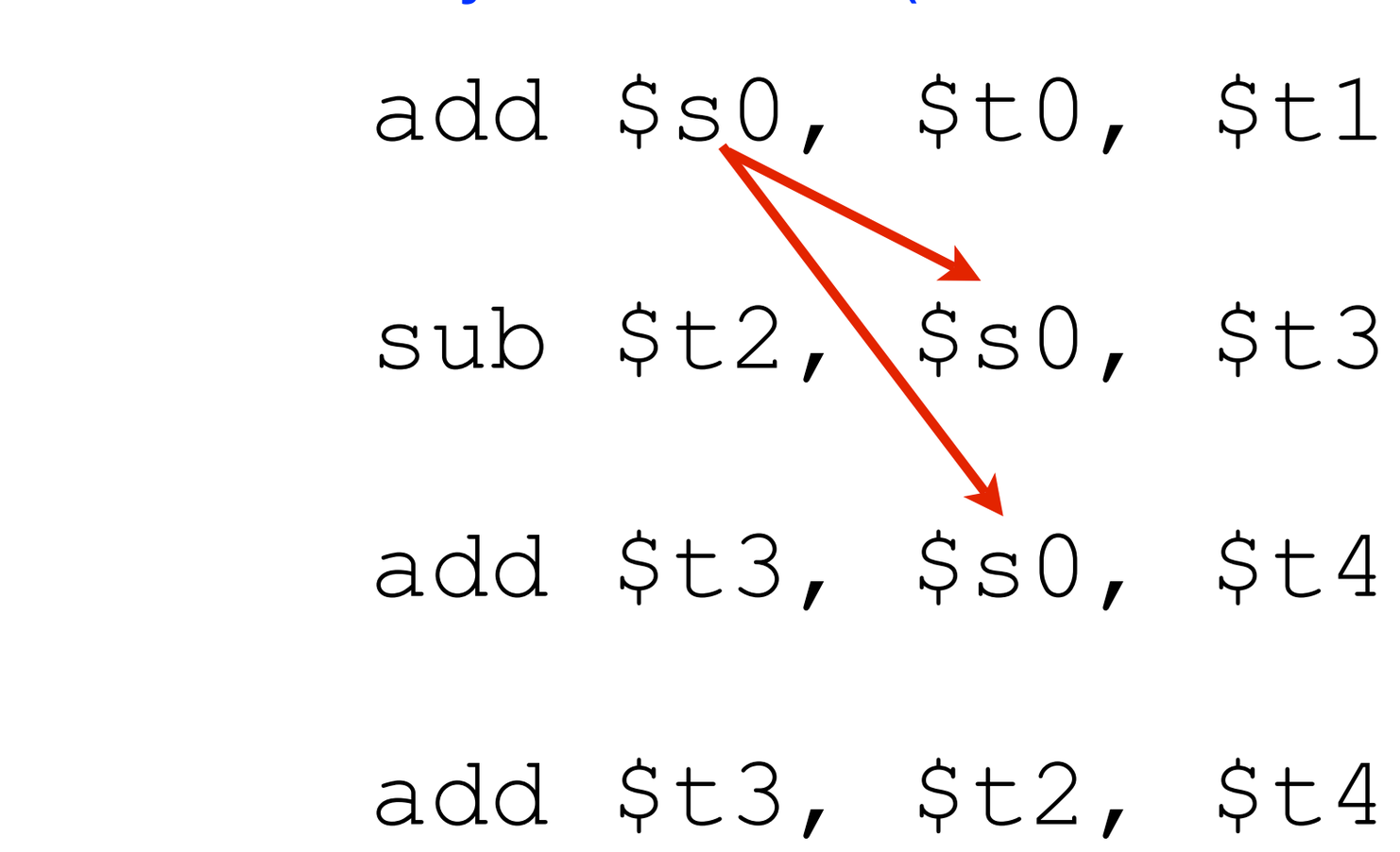
Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
  - Register values
  - Also memory accesses (more on this later)



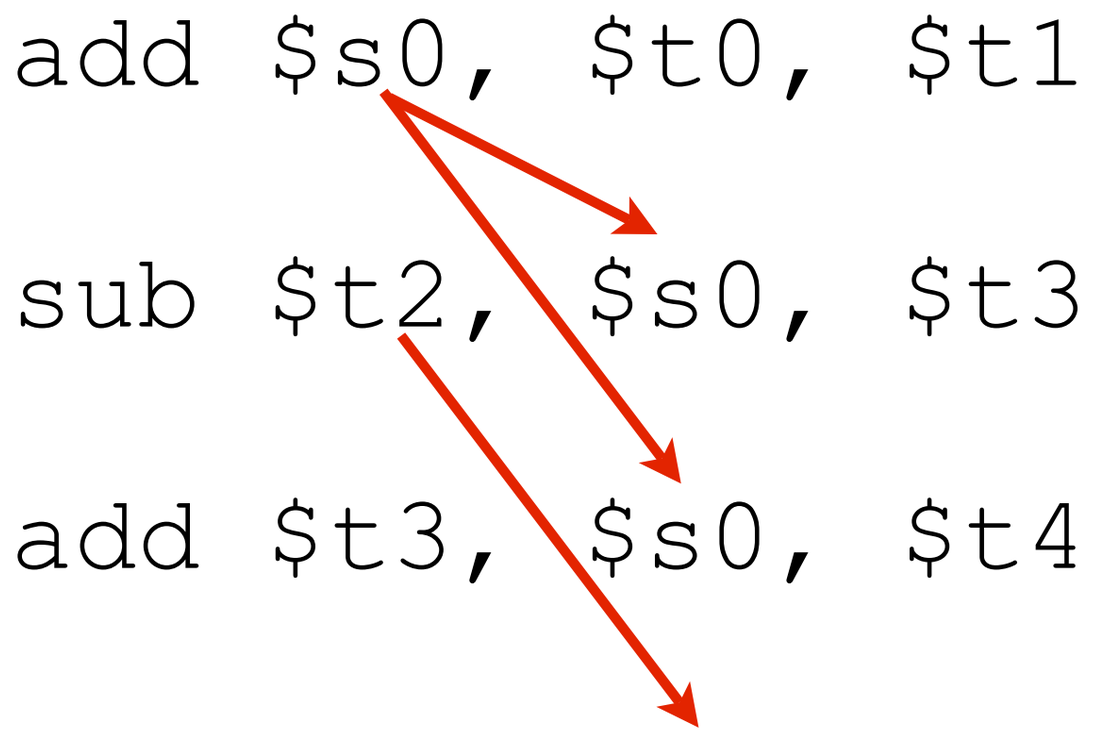
Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
  - Register values
  - Also memory accesses (more on this later)



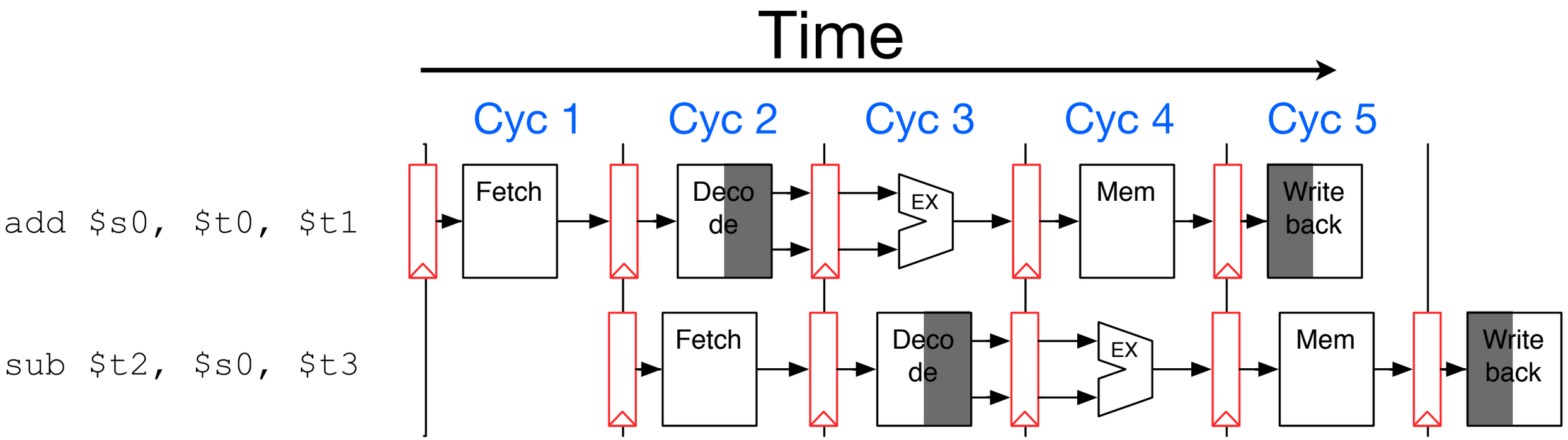
Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
  - Register values
  - Also memory accesses (more on this later)



Dependences in the pipeline

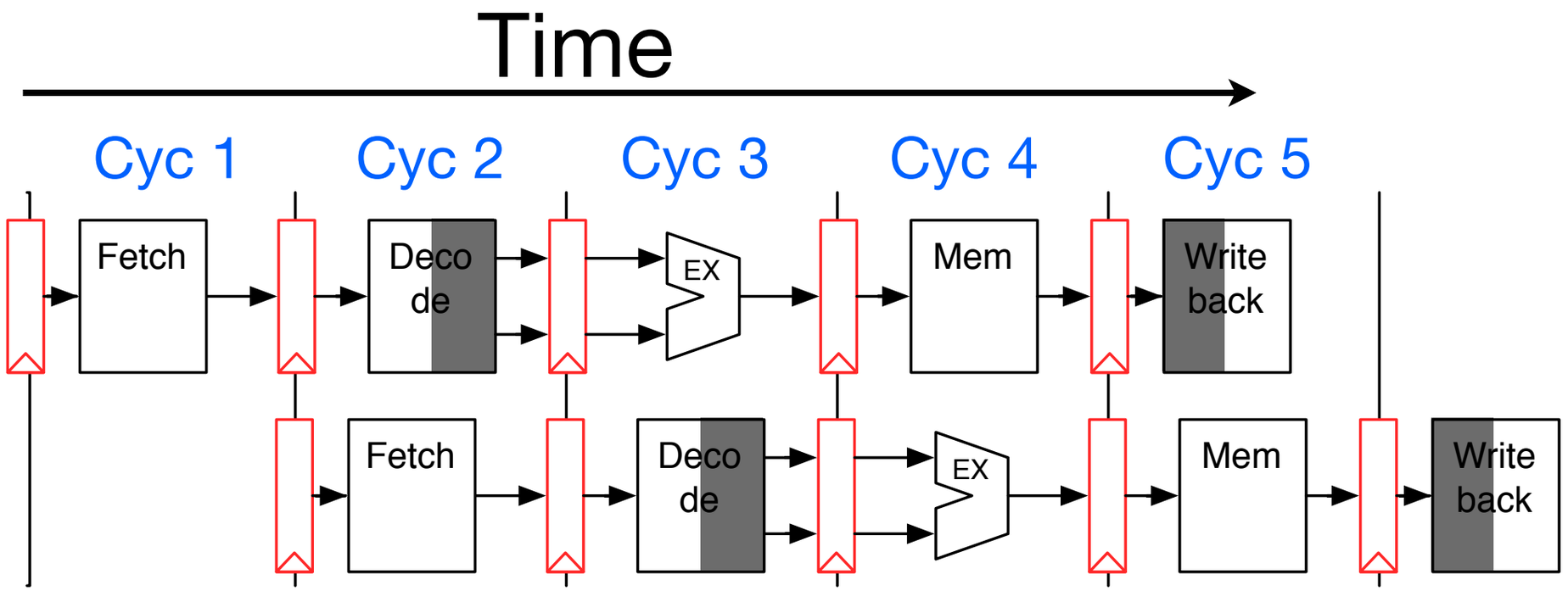
- In our simple pipeline, these instructions cause a data hazard



Dependences in the pipeline

- In our simple pipeline, these instructions cause a data hazard

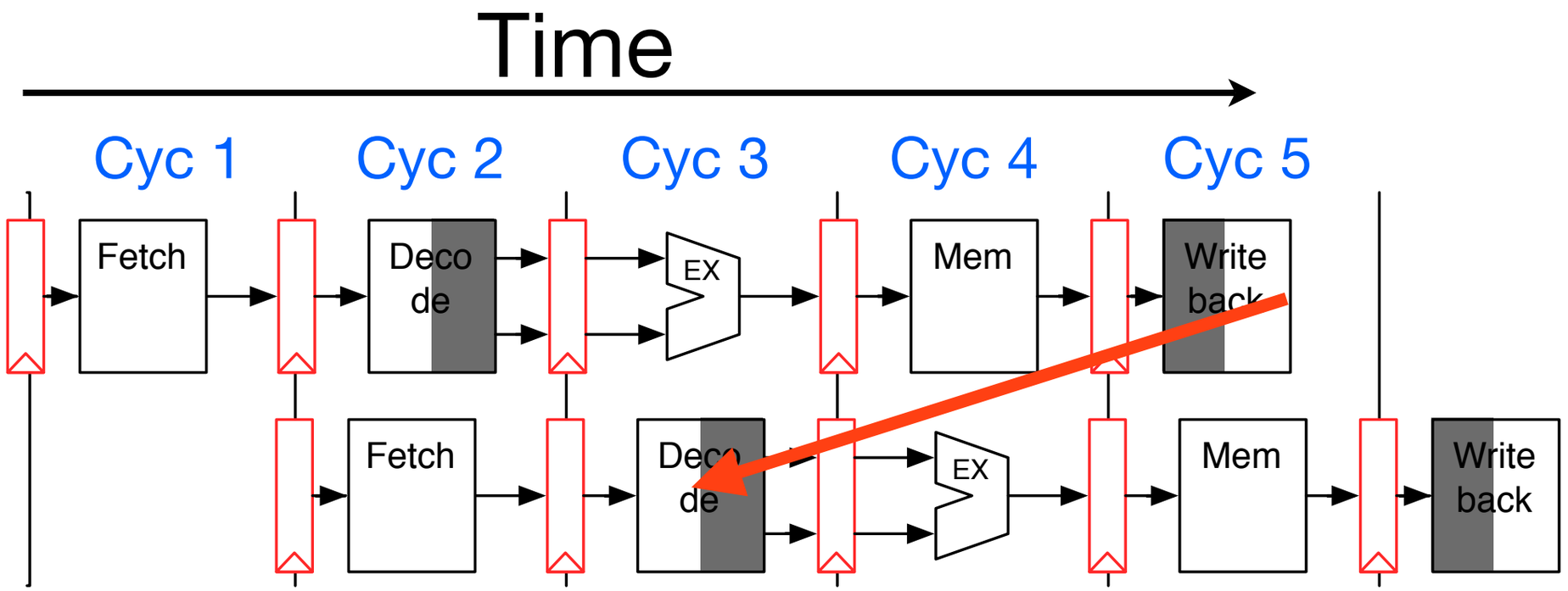
```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```



Dependences in the pipeline

- In our simple pipeline, these instructions cause a data hazard

```
add $s0, $t0, $t1

sub $t2, $s0, $t3
```

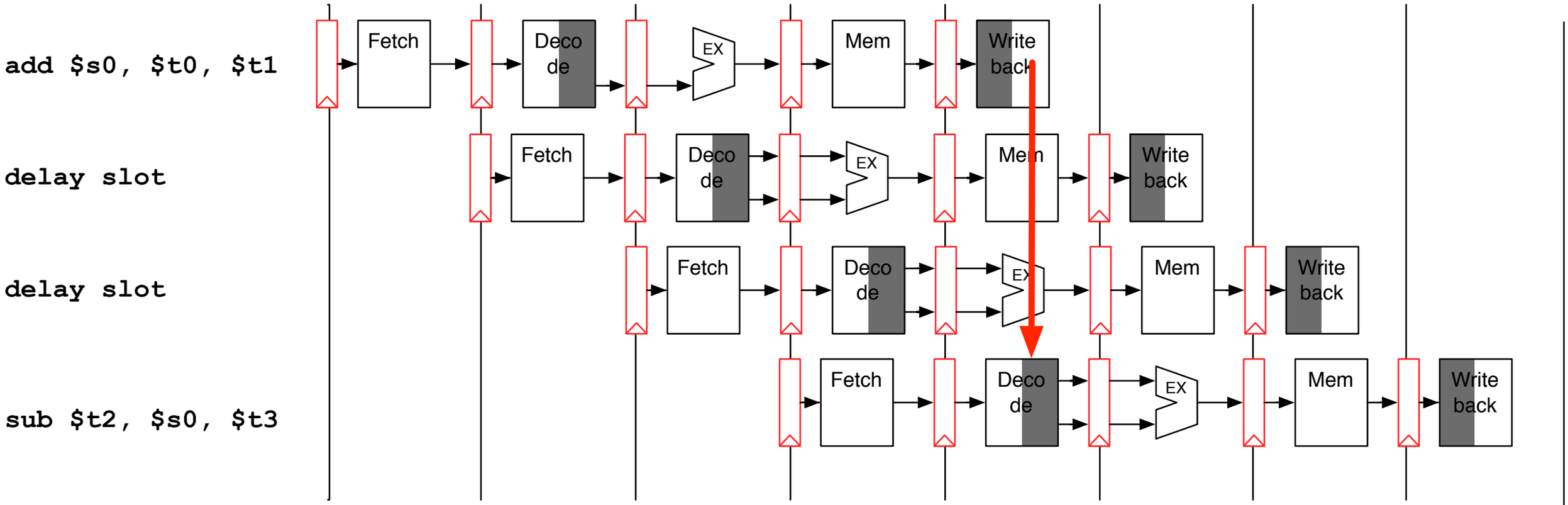


How can we fix it?

- Ideas?

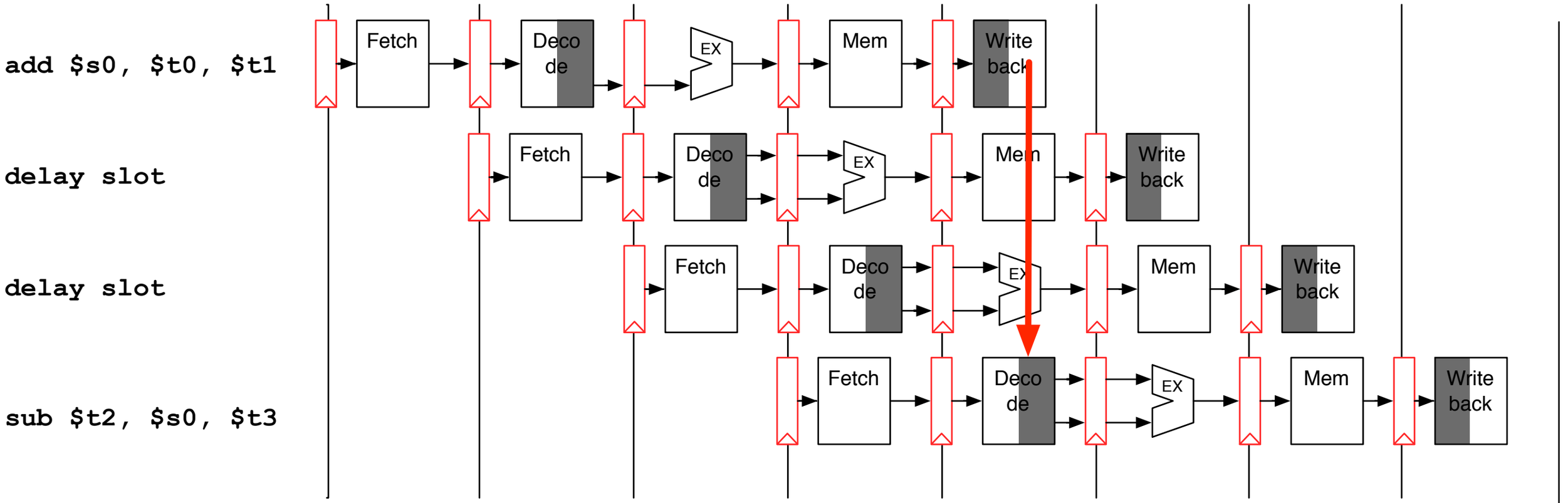
Solution 1: Make the compiler deal with it.

- Expose hazards to the big A architecture
  - A result is available N instructions after the instruction that generates it.
  - In the meantime, the register file has the old value.
  - This is called “a register delay slot”
- What is N? Can it change?



Solution 1: Make the compiler deal with it.

- Expose hazards to the big A architecture
  - A result is available N instructions after the instruction that generates it.
  - In the meantime, the register file has the old value.
  - This is called “a register delay slot”
- What is N? Can it change? N = 2, for our design



Compiling for delay slots

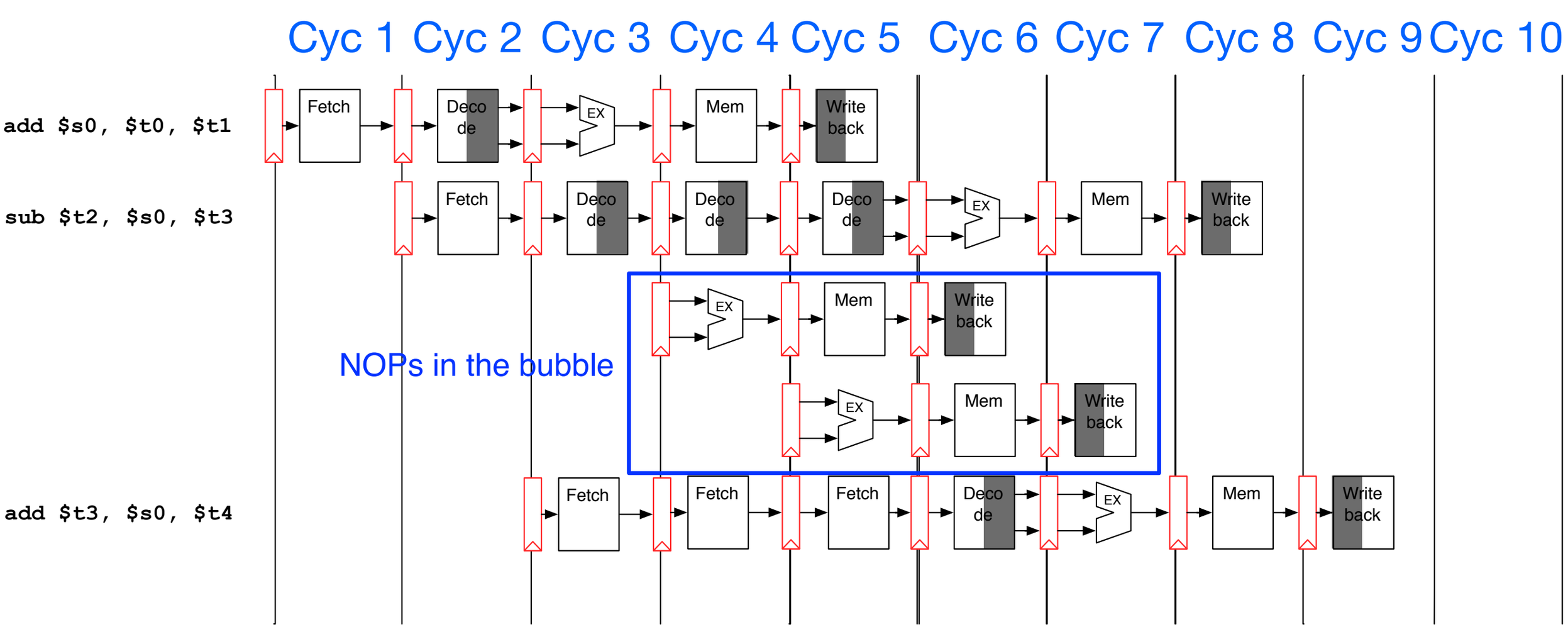
- The compiler must fill the delay slots
- Ideally, with useful instructions, but nops will work too.

```
add $s0, $t0, $t1

sub $t2, $s0, $t3

add $t3, $s0, $t4

and $t7, $t5, $t4
```

```
add $s0, $t0, $t1

and $t7, $t5, $t4

nop

sub $t2, $s0, $t3

add $t3, $s0, $t4
```

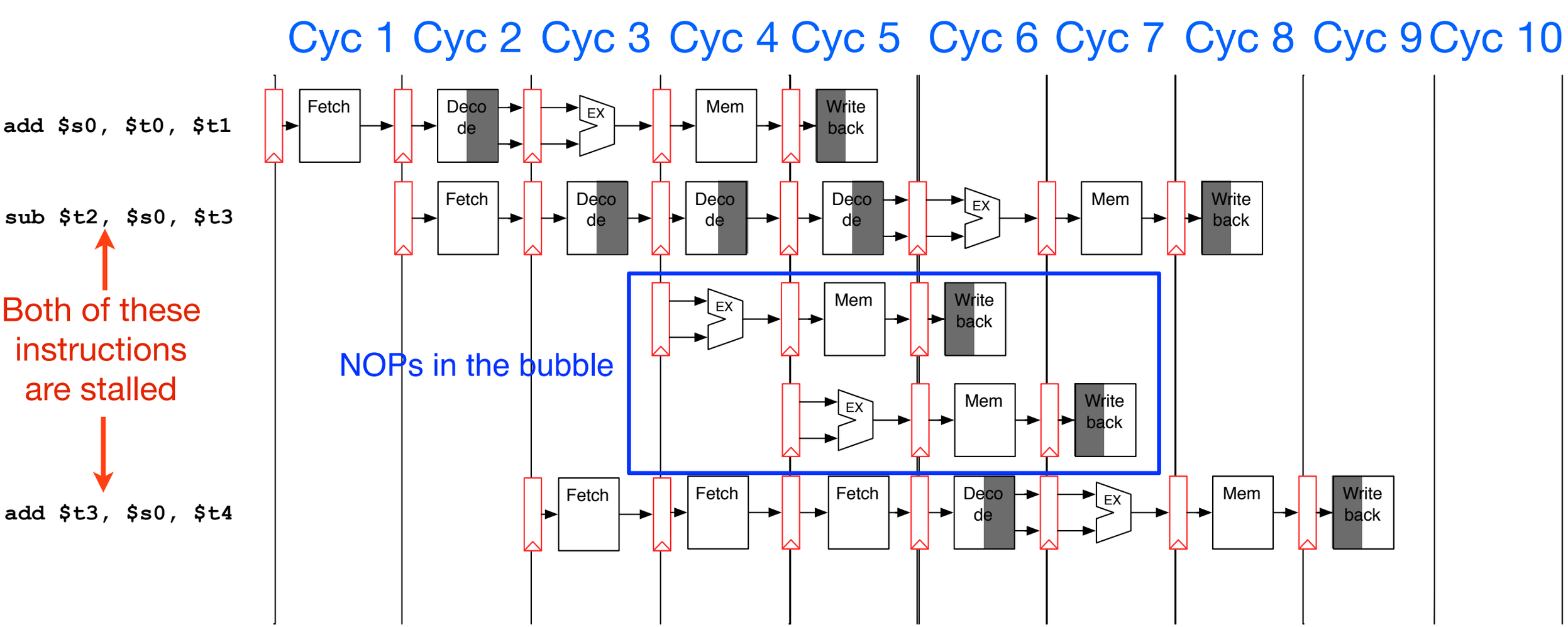
Solution 2: Stall

- When you need a value that is not ready, “stall”
  - Suspend the execution of the executing instruction
  - and those that follow.
  - This introduces a pipeline “bubble.”
- A bubble is a lack of work to do, it propagates through the pipeline like nop instructions



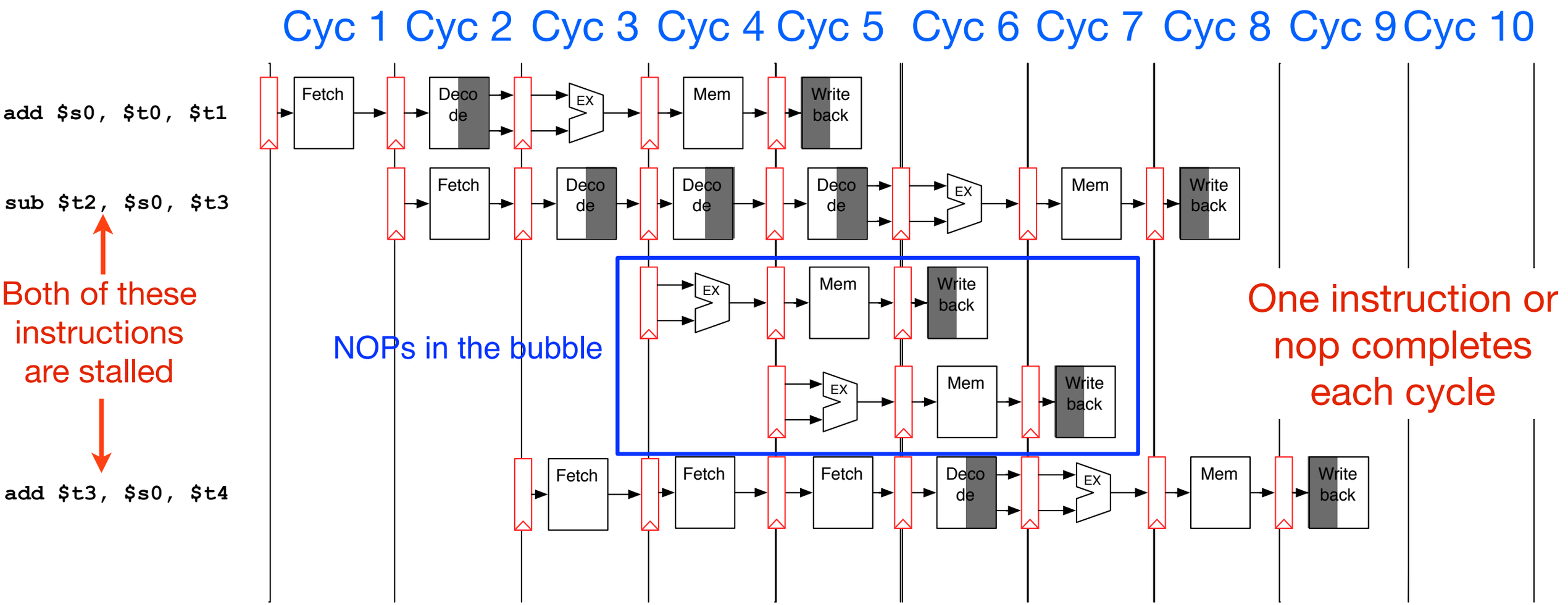
Solution 2: Stall

- When you need a value that is not ready, “stall”
  - Suspend the execution of the executing instruction
  - and those that follow.
  - This introduces a pipeline “bubble.”
- A bubble is a lack of work to do, it propagates through the pipeline like nop instructions



Solution 2: Stall

- When you need a value that is not ready, “stall”
  - Suspend the execution of the executing instruction
  - and those that follow.
  - This introduces a pipeline “bubble.”
- A bubble is a lack of work to do, it propagates through the pipeline like nop instructions



Stalling the pipeline

- Freeze all pipeline stages before the stage where the hazard occurred.
  - Disable the PC update
  - Disable the pipeline registers
- This is equivalent to inserting into the pipeline when a hazard exists
  - Insert nop control bits at stalled stage (decode in our example)
  - How is this solution still potentially “better” than relying on the compiler?

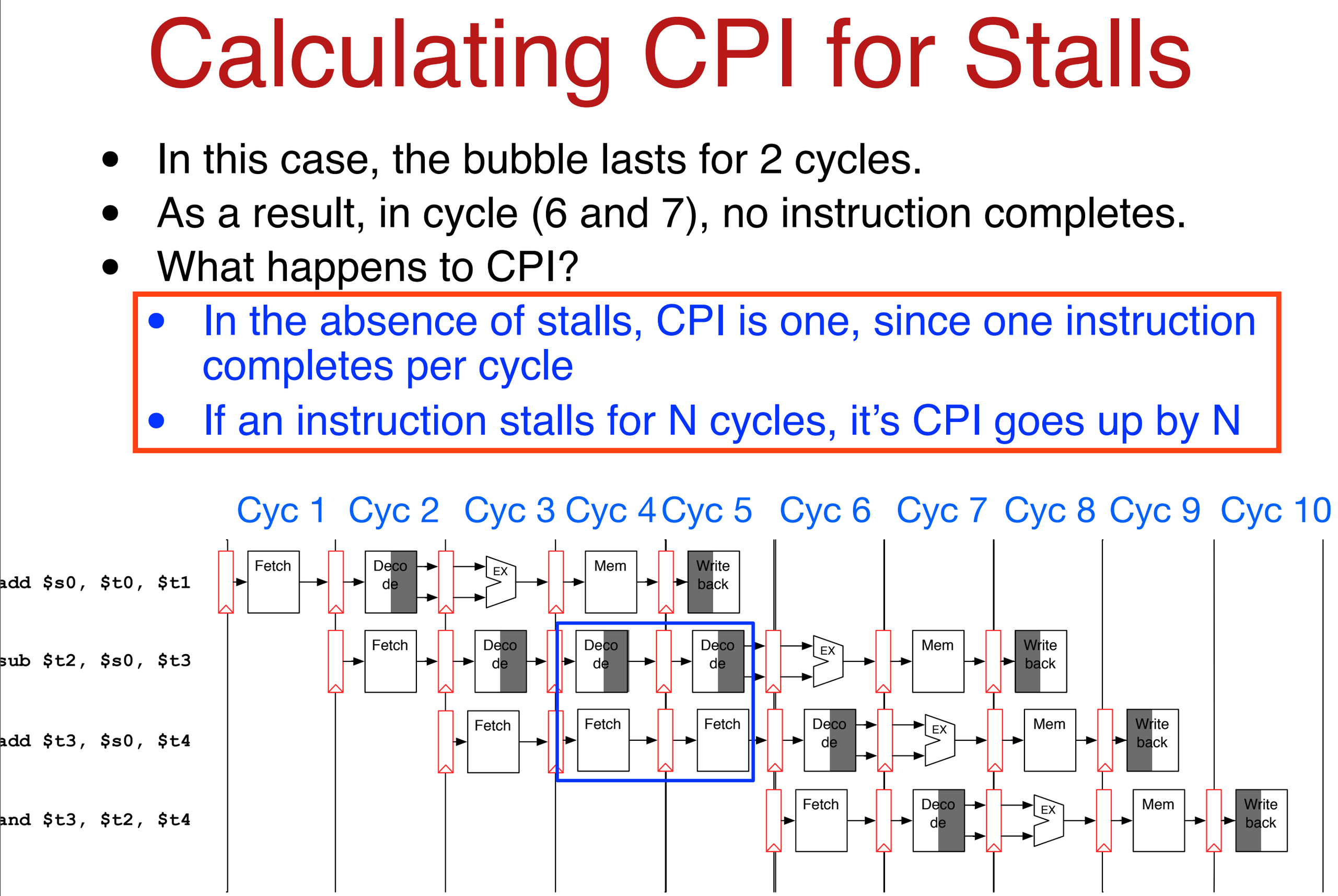
Stalling the pipeline

- Freeze all pipeline stages before the stage where the hazard occurred.
  - Disable the PC update
  - Disable the pipeline registers
- This is equivalent to inserting into the pipeline when a hazard exists
  - Insert nop control bits at stalled stage (decode in our example)
  - How is this solution still potentially “better” than relying on the compiler?

The compiler can still act like there are delay slots to avoid stalls.
Implementation details are not exposed in the ISA

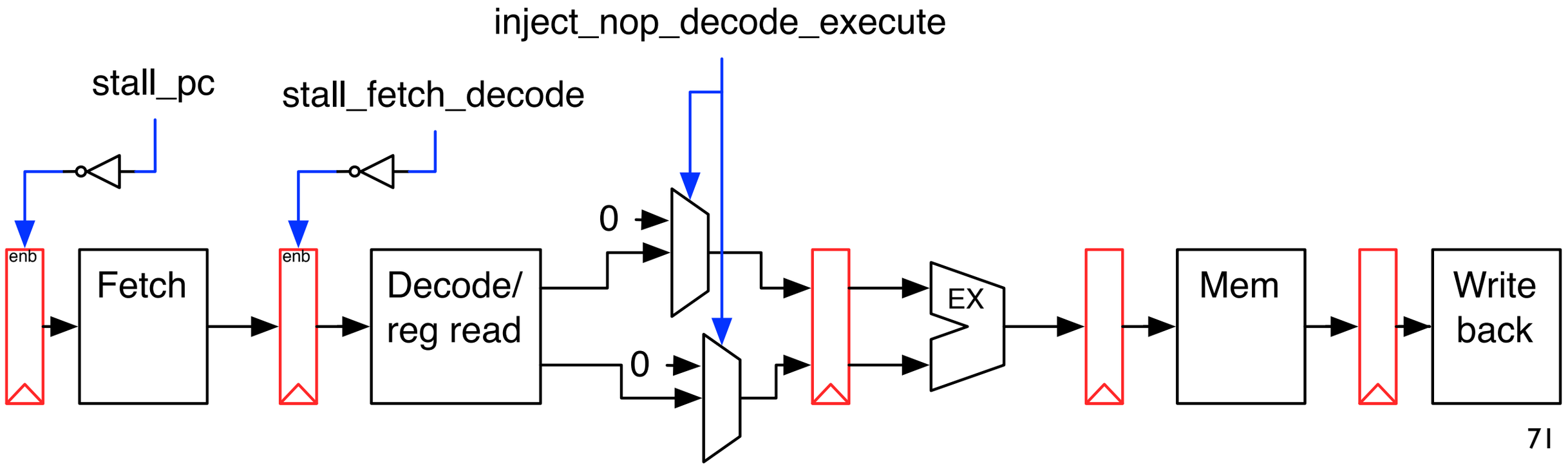
Calculating CPI for Stalls

- In this case, the bubble lasts for 2 cycles.
- As a result, in cycle (6 and 7), no instruction completes.
- What happens to CPI?
  - In the absence of stalls, CPI is one, since one instruction completes per cycle
  - If an instruction stalls for N cycles, it's CPI goes up by N



Hardware for Stalling

- Turn off the enables on the earlier pipeline stages
  - The earlier stages will keep processing the same instruction over and over.
  - No new instructions get fetched.
- Insert control and data values corresponding to a nop into the "downstream" pipeline register.
  - This will create the bubble.
  - The nops will flow downstream, doing nothing.
- When the stall is over, re-enable the pipeline registers
  - The instructions in the "upstream" stages will start moving again.
  - New instructions will start entering the pipeline again.



The Impact of Stalling On Performance

- ET = I * CPI * CT
- I and CT are constant
- What is the impact of stalling on CPI?

- What do we need to know to figure it out?

The Impact of Stalling On Performance

- ET = I * CPI * CT
- I and CT are constant
- What is the impact of stalling on CPI?

- Fraction of instructions that stall: 30%
- Baseline CPI = 1
- Stall CPI = 1 + 2 = 3

- New CPI =

The Impact of Stalling On Performance

- ET = I * CPI * CT
- I and CT are constant
- What is the impact of stalling on CPI?

- Fraction of instructions that stall: 30%
- Baseline CPI = 1
- Stall CPI = 1 + 2 = 3

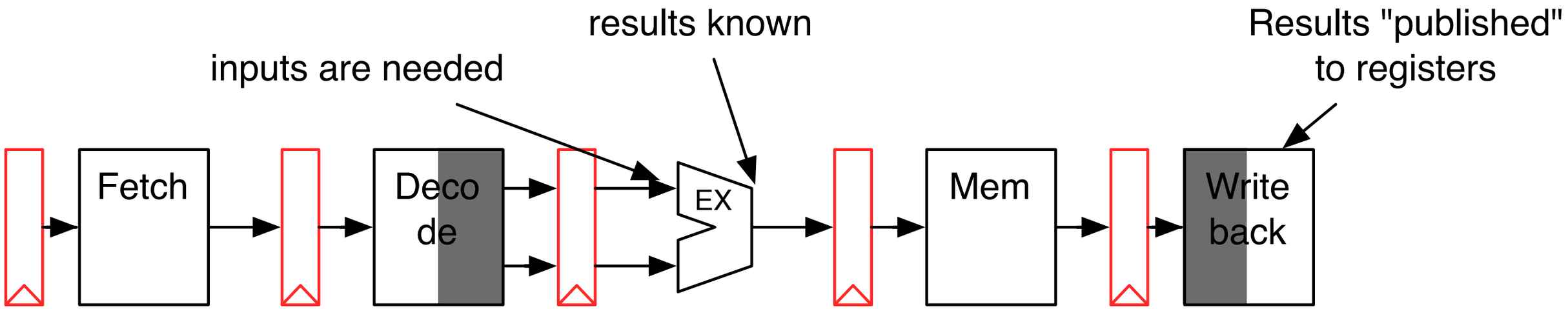
- New CPI =

0.3*3 + 0.7*1 = 1.6

Solution 3: Bypassing/Forwarding

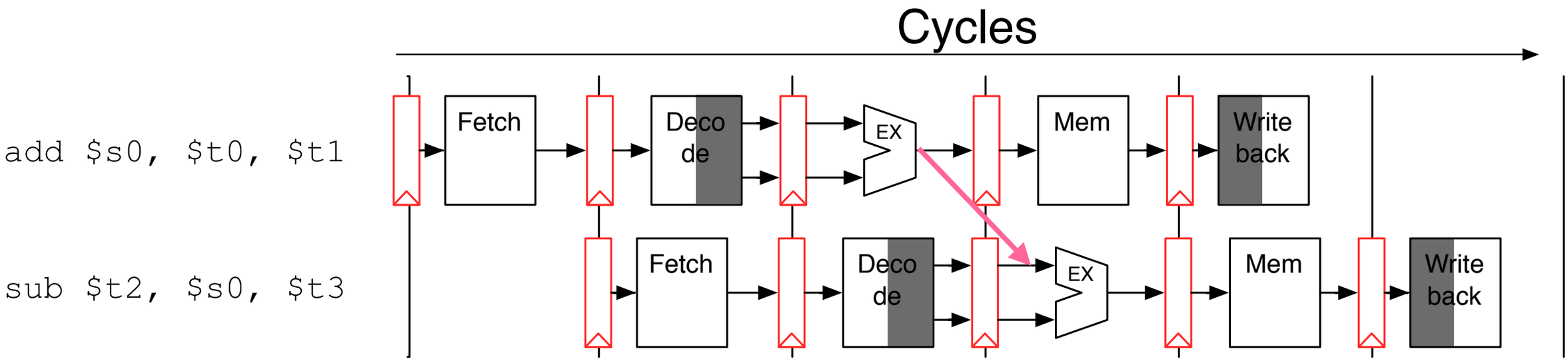
- Data values are computed in Ex and Mem but “publicized in write back”

- The data exists! We should use it.



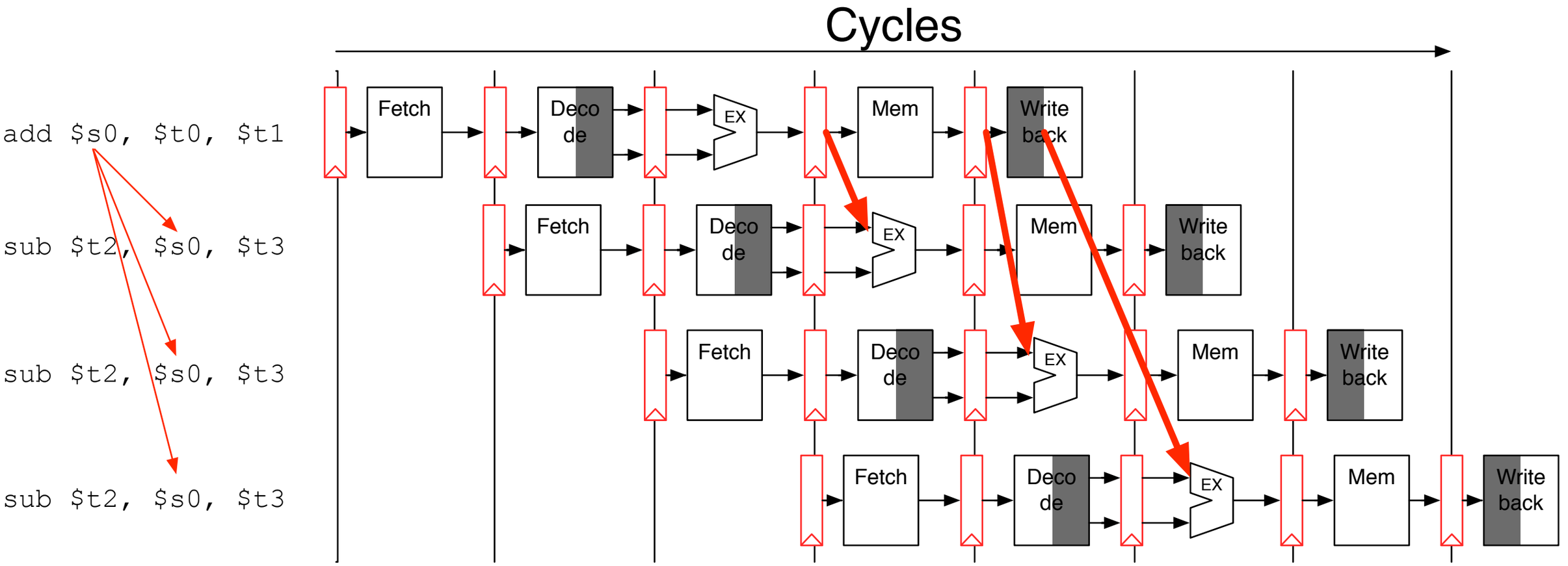
Bypassing or Forwarding

- Take the values, where ever they are

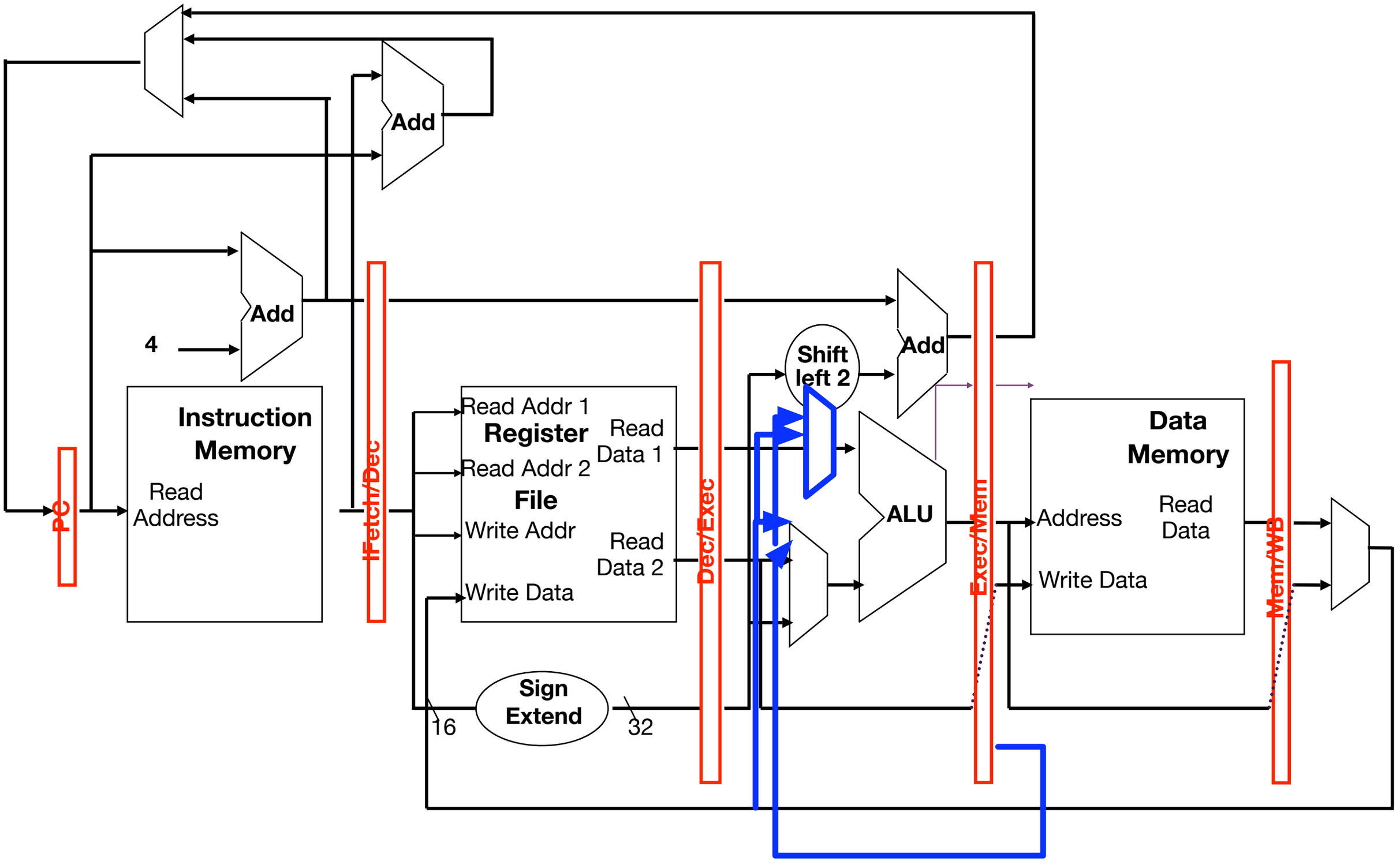


-

Forwarding Paths



Forwarding in Hardware

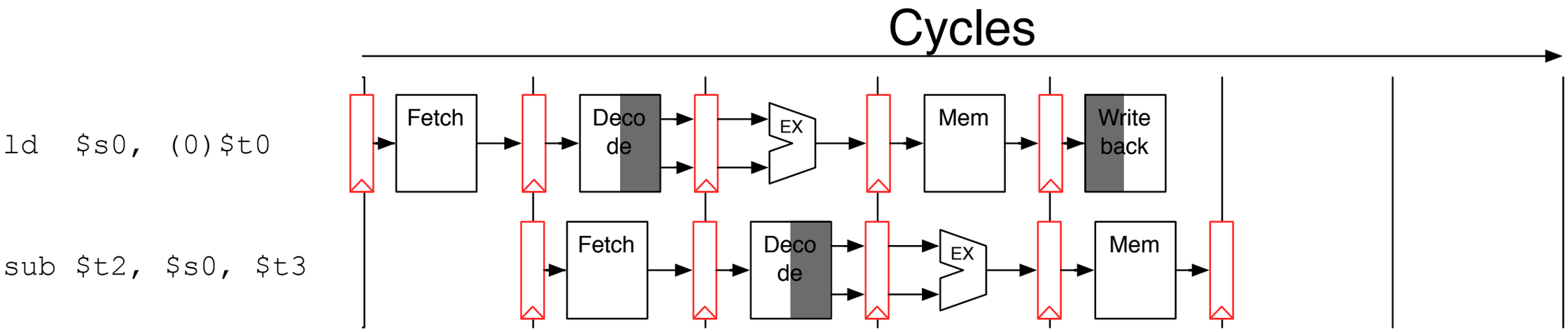


Hardware Cost of Forwarding

- In our pipeline, adding forwarding required relatively little hardware.
- For deeper pipelines it gets much more expensive
  - Roughly: ALU * pipe_stages you need to forward over
  - Some modern processor have multiple ALUs (4-5)
  - And deeper pipelines (4-5 stages of to forward across)
- Not all forwarding paths need to be supported.
  - If a path does not exist, the processor will need to stall.

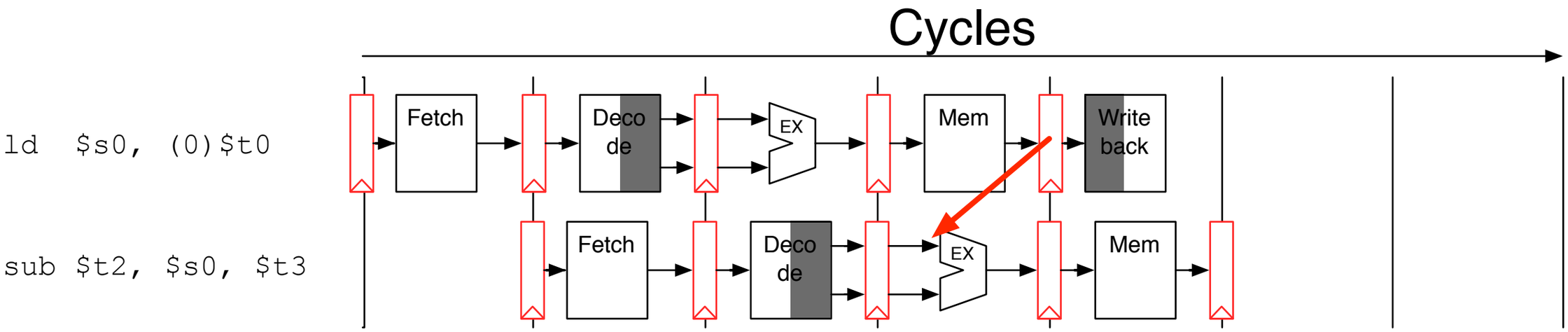
Forwarding for Loads

- Values can also come from the Mem stage
  - In this case, forward is not possible to the next instruction (and is not necessary for later instructions)
- Choices
  - Always stall!
  - Stall when needed!
  - Expose this in the ISA.



Forwarding for Loads

- Values can also come from the Mem stage
  - In this case, forward is not possible to the next instruction (and is not necessary for later instructions)
- Choices
  - Always stall!
  - Stall when needed!
  - Expose this in the ISA.

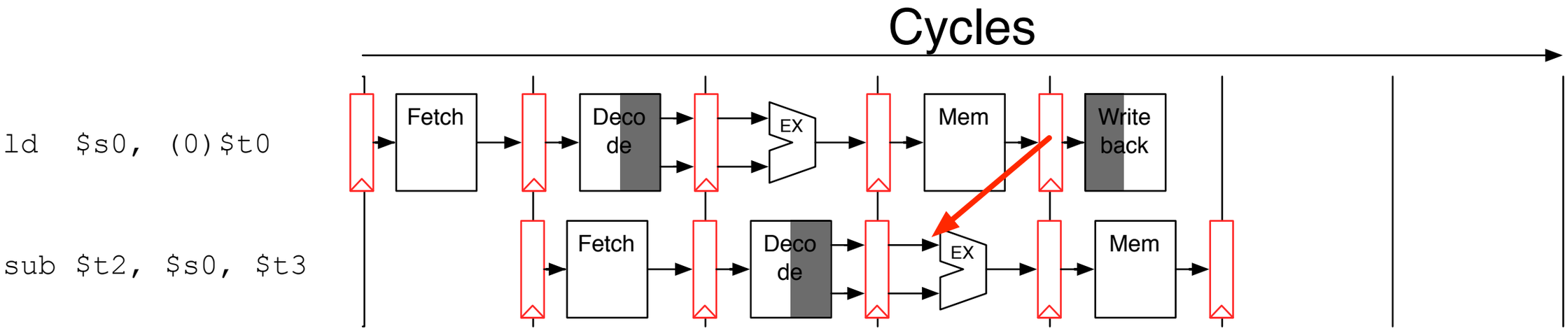


Time travel presents significant implementation challenges

Forwarding for Loads

- Values can also come from the Mem stage
  - In this case, forward is not possible to the next instruction (and is not necessary for later instructions)
- Choices
  - Always stall!
  - Stall when needed!
  - Expose this in the ISA.

Which does MIPS do?

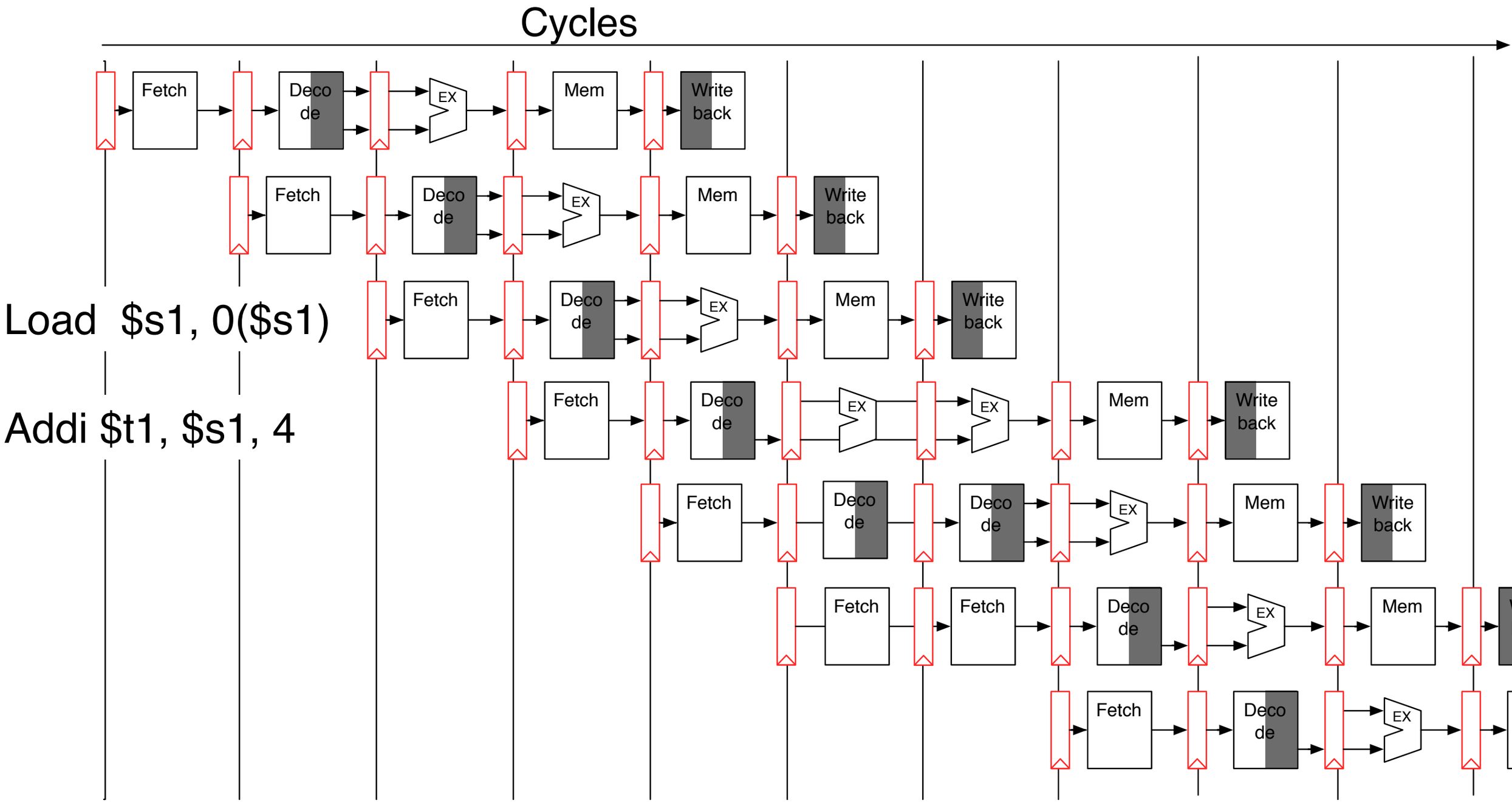


Time travel presents significant implementation challenges

Pros and Cons

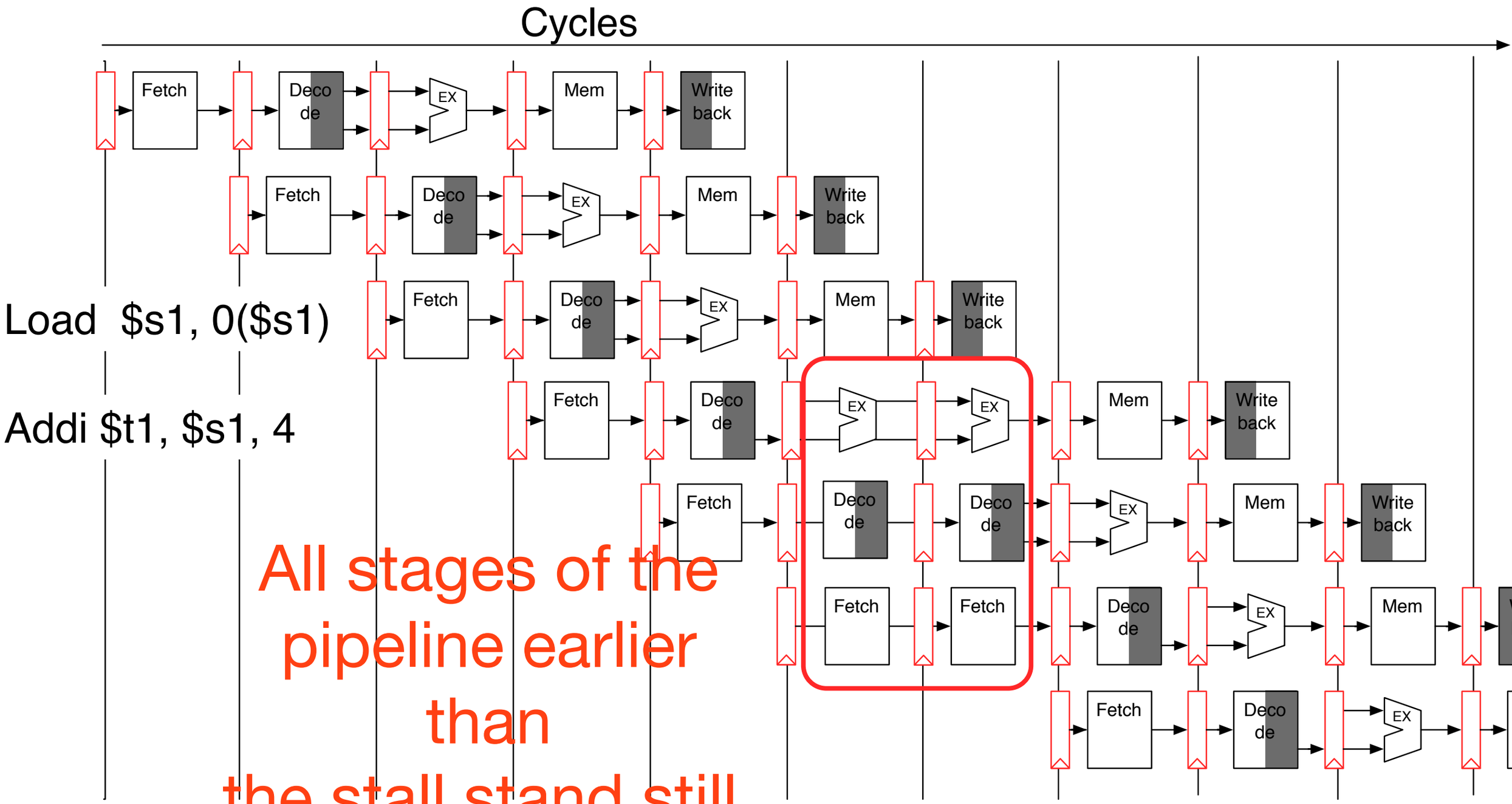
- Punt to the compiler
  - This is what MIPS does and is the source of the load-delay slot
  - Future versions must emulate a single load-delay slot.
  - The compiler fills the slot if possible, or drops in a nop.
- Always stall.
  - The compiler is oblivious, but performance will suffer
  - 10-15% of instructions are loads, and the CPI for loads will be 2
- Forward when possible, stall otherwise
  - Here the compiler can order instructions to avoid the stall.
  - If the compiler can’t fix it, the hardware will.

Stalling for Load



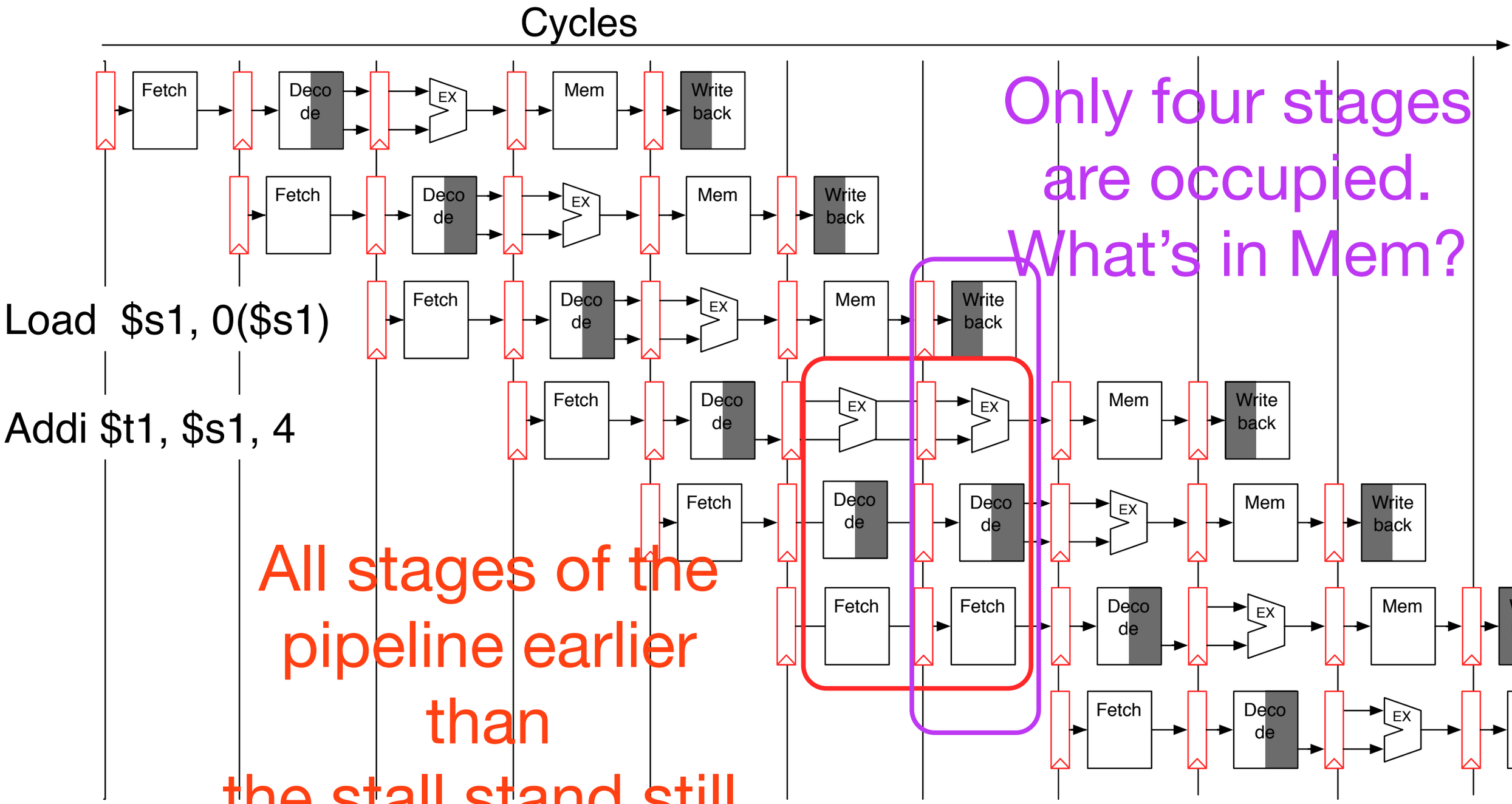
To “stall” we insert a noop *in place of* the instruction and freeze the earlier stages of the pipeline

Stalling for Load



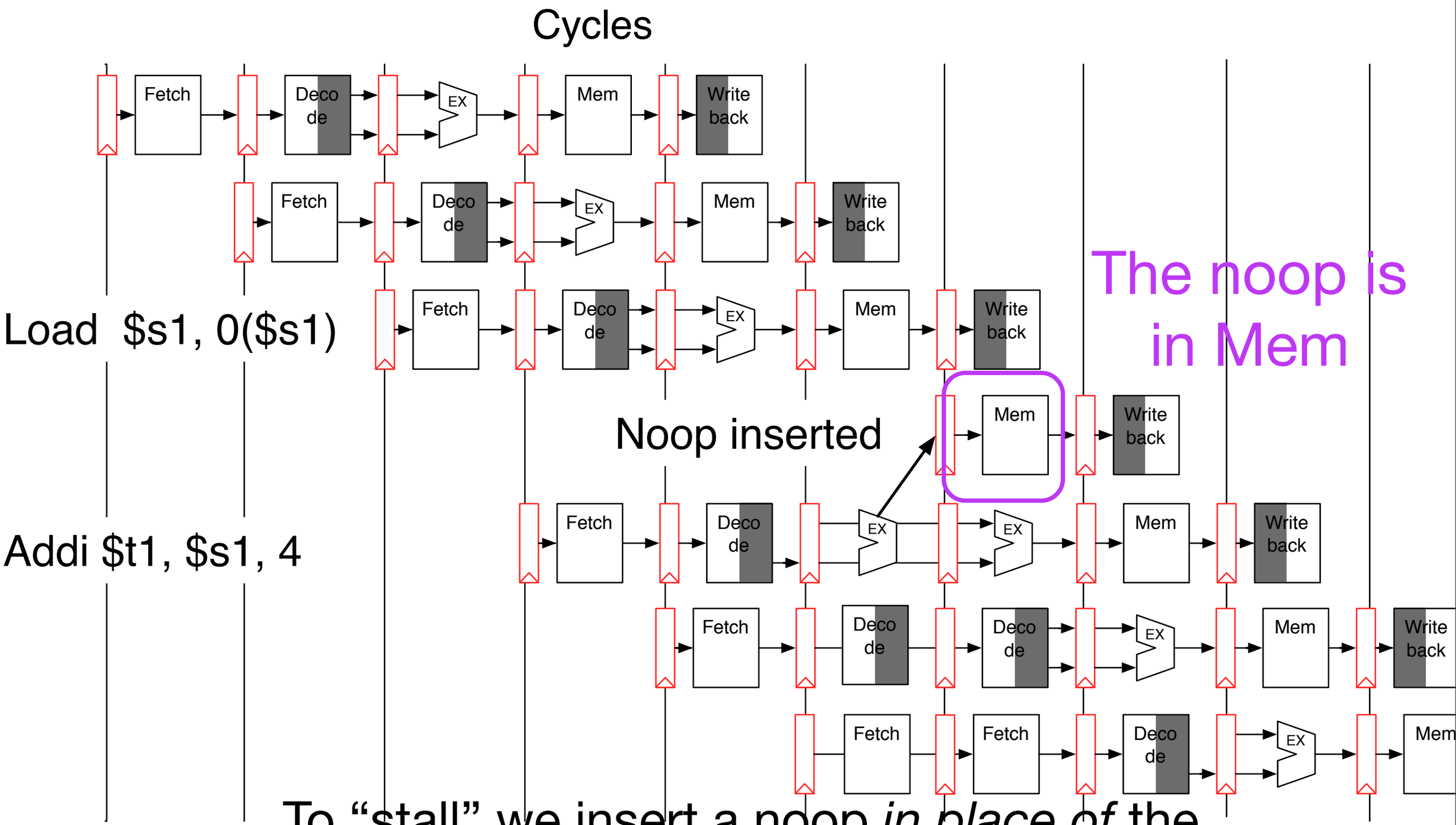
To “stall” we insert a noop *in place of* the instruction and freeze the earlier stages of the pipeline

Stalling for Load



To "stall" we insert a noop *in place of* the instruction and freeze the earlier stages of the pipeline

Inserting Noops



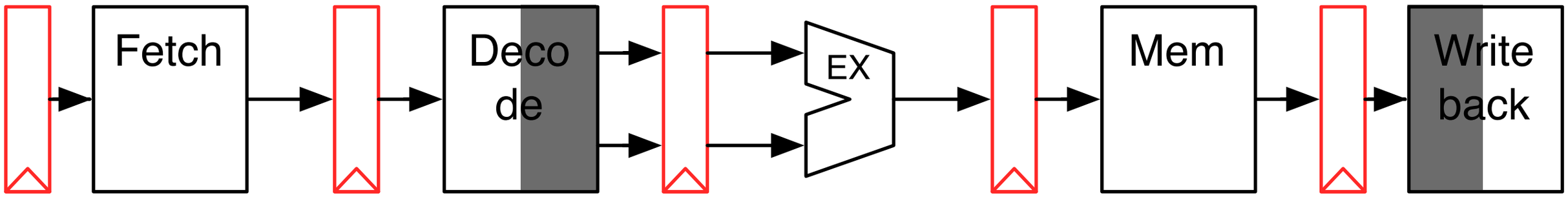
To “stall” we insert a noop *in place of* the instruction and freeze the earlier stages of the pipeline

Key Points: Control Hazards

- Control occur when we don't know what the next instruction is
- Caused by branches and jumps.
- Strategies for dealing with them
  - Stall
  - Guess!
    - Leads to speculation
    - Flushing the pipeline
    - Strategies for making better guesses
- Understand the difference between stall and flush

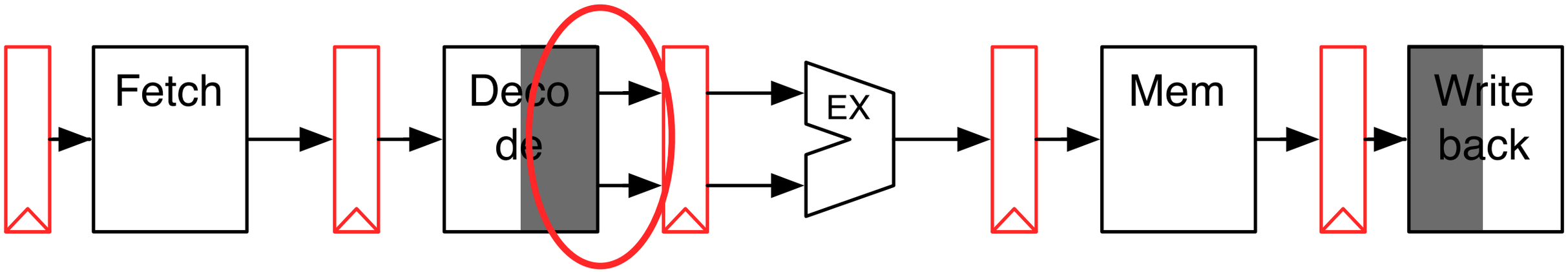
Computing the PC Normally

- Non-branch instruction
  - PC = PC + 4
- When is PC ready?



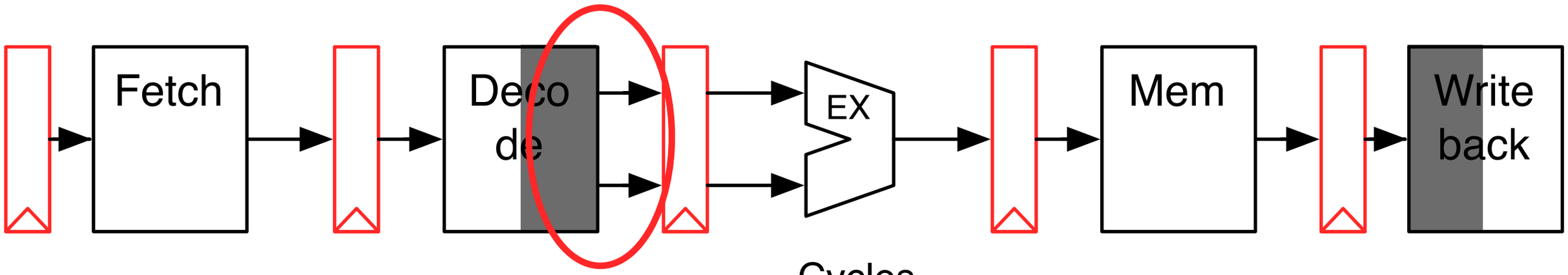
Computing the PC Normally

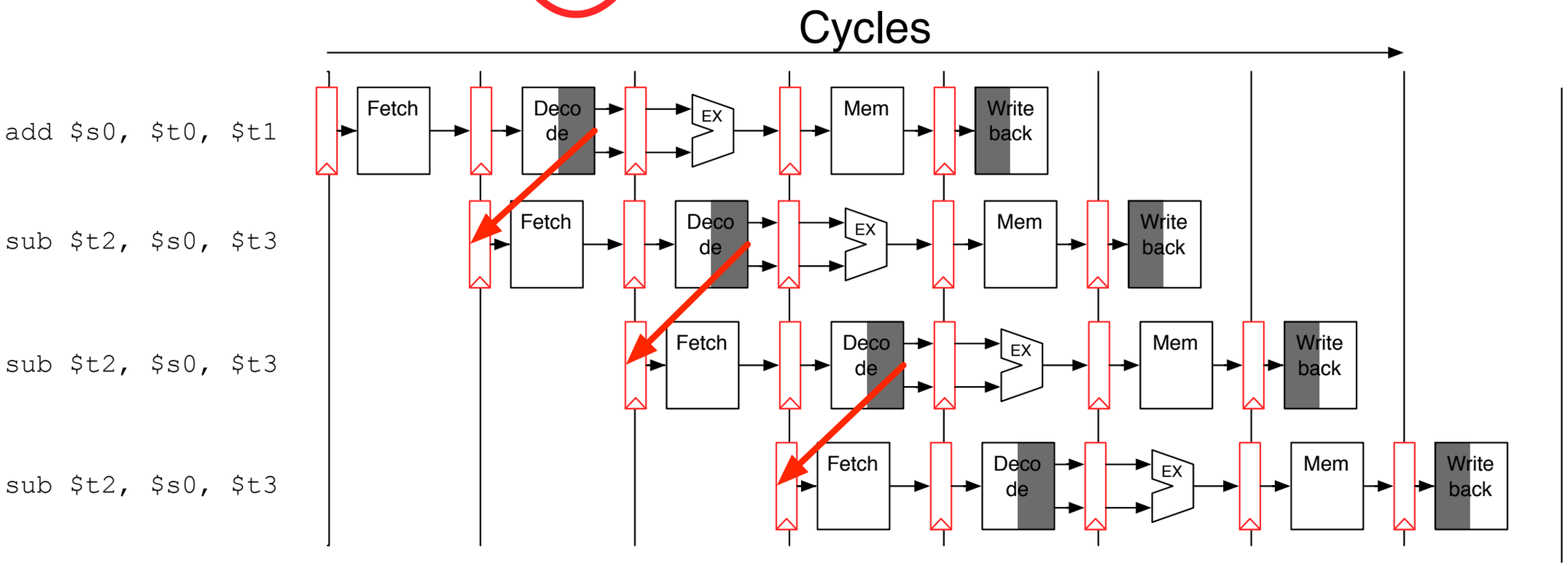
- Non-branch instruction
  - PC = PC + 4
- When is PC ready?



Computing the PC Normally

- Non-branch instruction
  - PC = PC + 4
- When is PC ready?



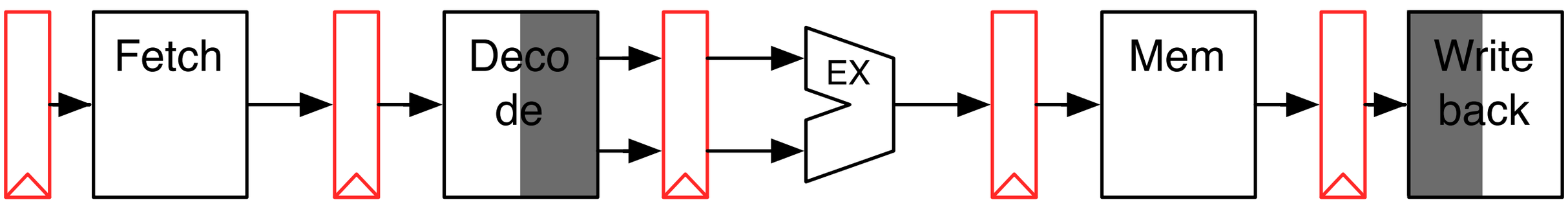


Fixing the Ubiquitous Control Hazard

- We need to know if an instruction is a branch in the fetch stage!
- How can we accomplish this?

Solution 1: Partially decode the instruction in fetch. You just need to know if it's a branch, a jump, or something else.

Solution 2: We'll discuss later.

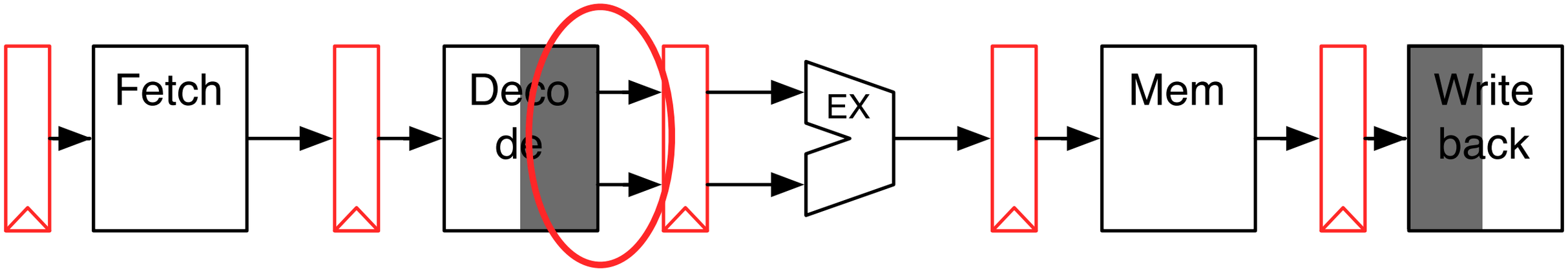


Fixing the Ubiquitous Control Hazard

- We need to know if an instruction is a branch in the fetch stage!
- How can we accomplish this?

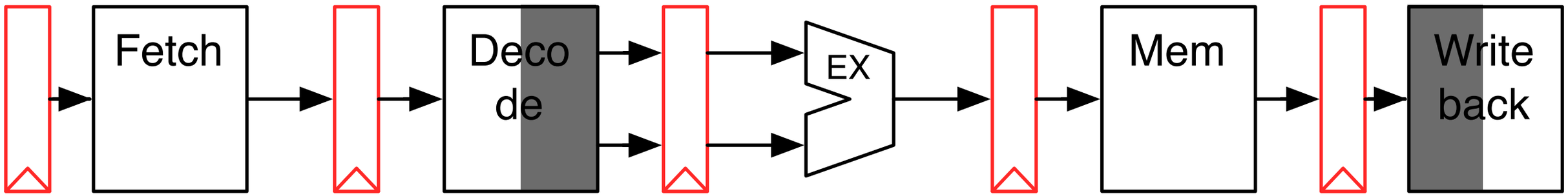
Solution 1: Partially decode the instruction in fetch. You just need to know if it's a branch, a jump, or something else.

Solution 2: We'll discuss later.



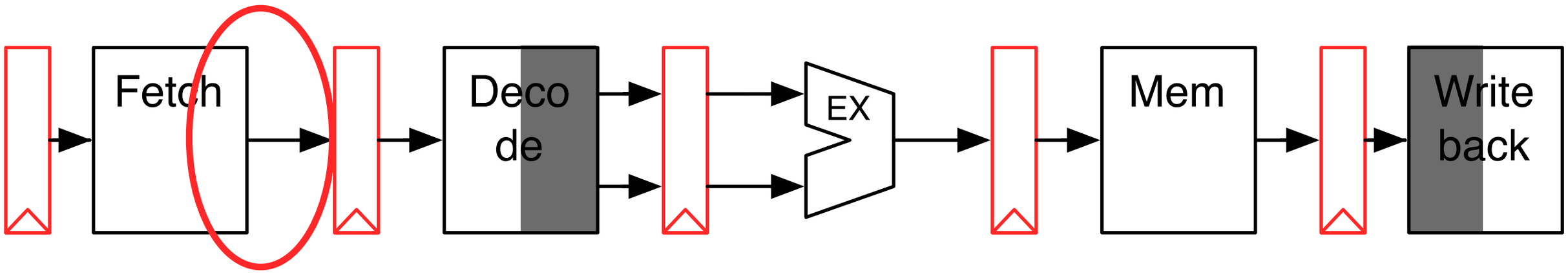
Computing the PC Normally

- Pre-decode in the fetch unit.
  - PC = PC + 4
- The PC is ready for the next fetch cycle.



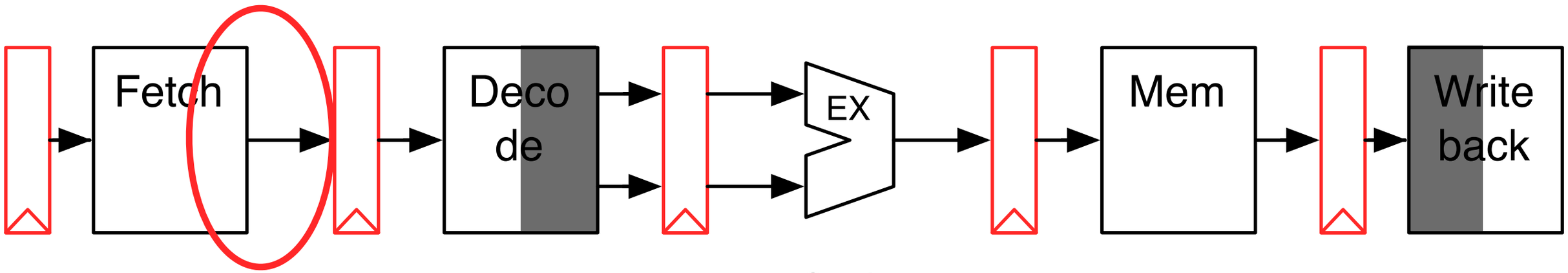
Computing the PC Normally

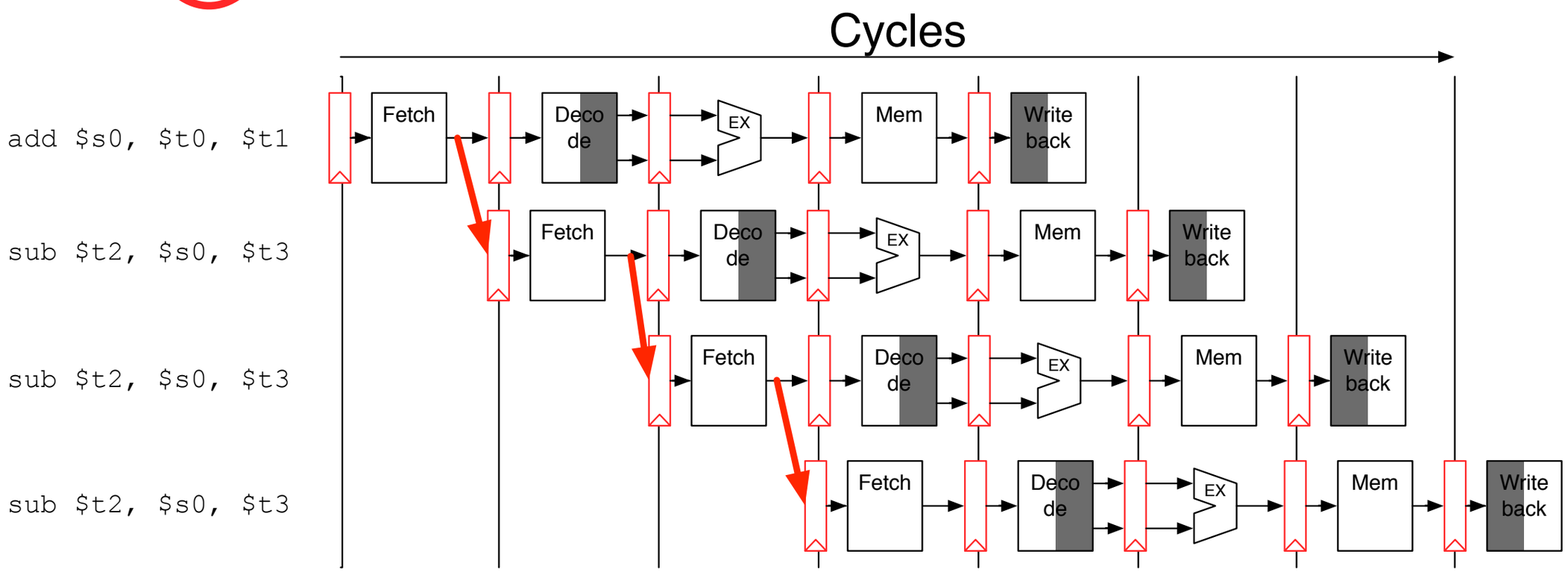
- Pre-decode in the fetch unit.
  - PC = PC + 4
- The PC is ready for the next fetch cycle.



Computing the PC Normally

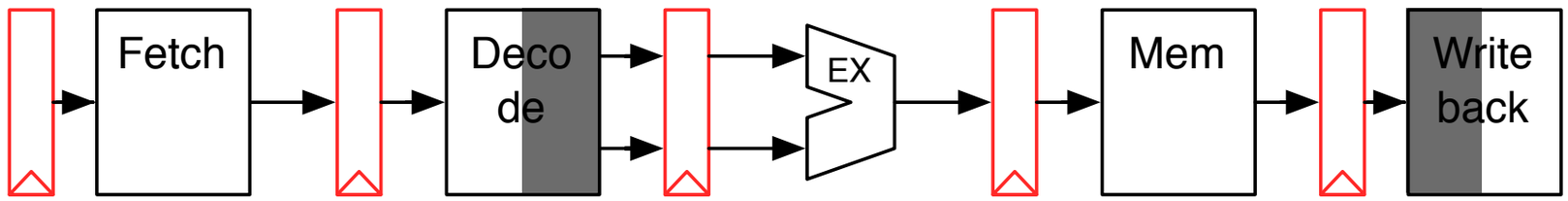
- Pre-decode in the fetch unit.
  - PC = PC + 4
- The PC is ready for the next fetch cycle.

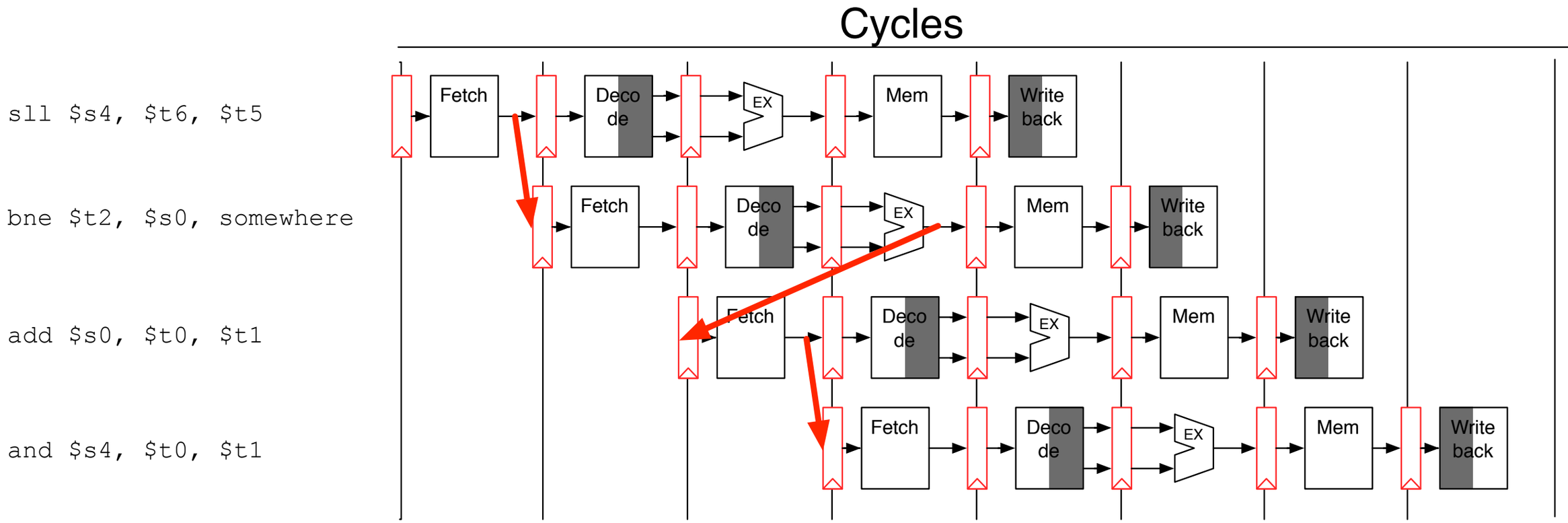




Computing the PC for Branches

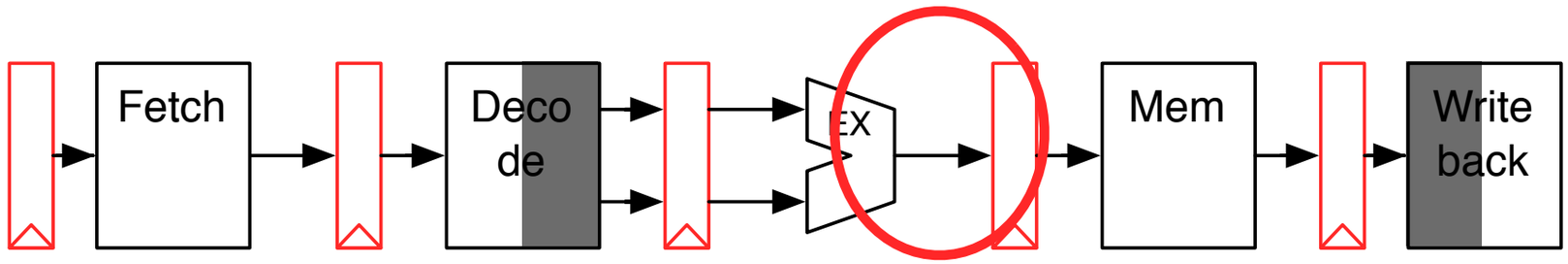
- Branch instructions
  - bne $s1, $s2, offset
  - if ($s1 != $s2) { PC = PC + offset} else {PC = PC + 4;}
- When is the value ready?

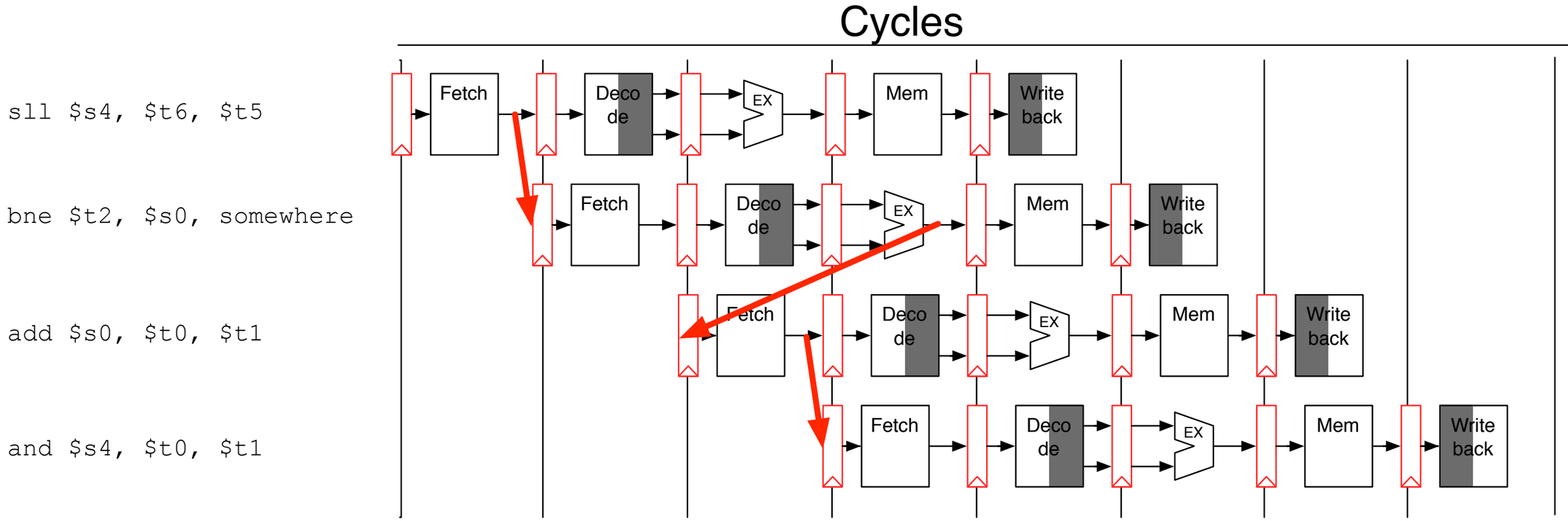




Computing the PC for Branches

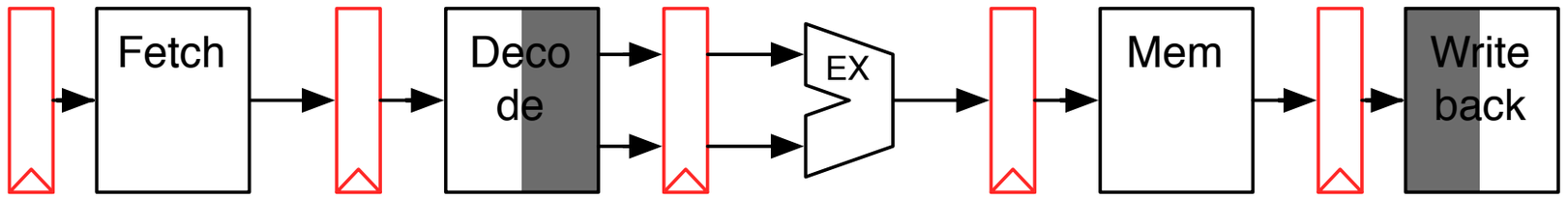
- Branch instructions
  - bne $s1, $s2, offset
  - if ($s1 != $s2) { PC = PC + offset} else {PC = PC + 4;}
- When is the value ready?

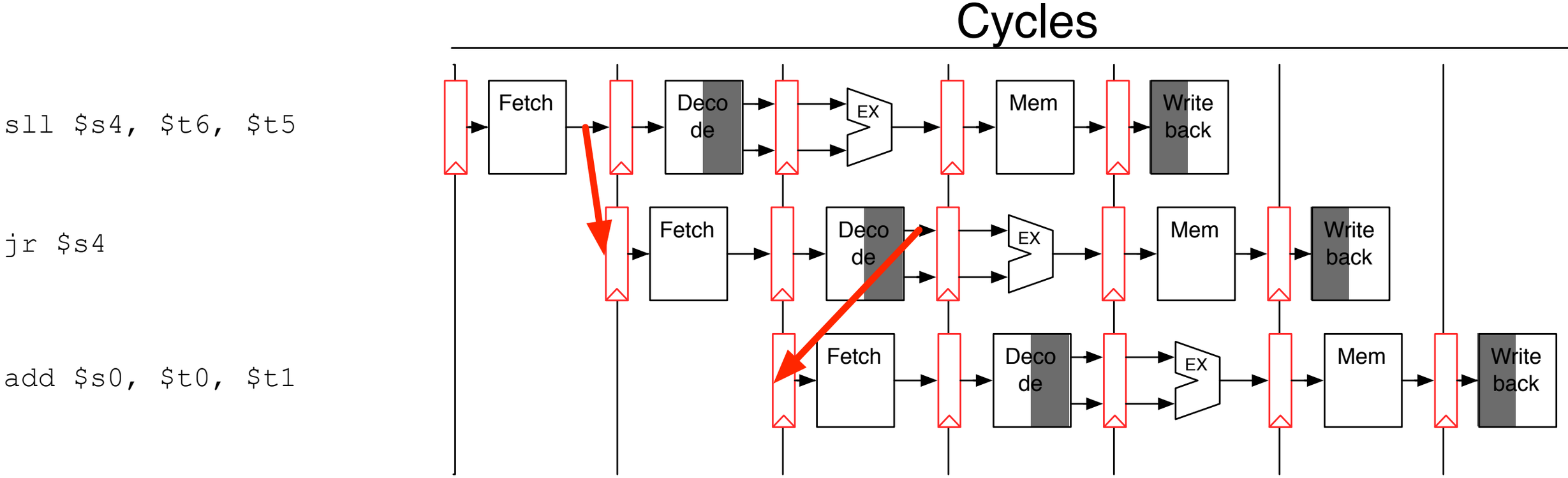




Computing the PC for Jumps

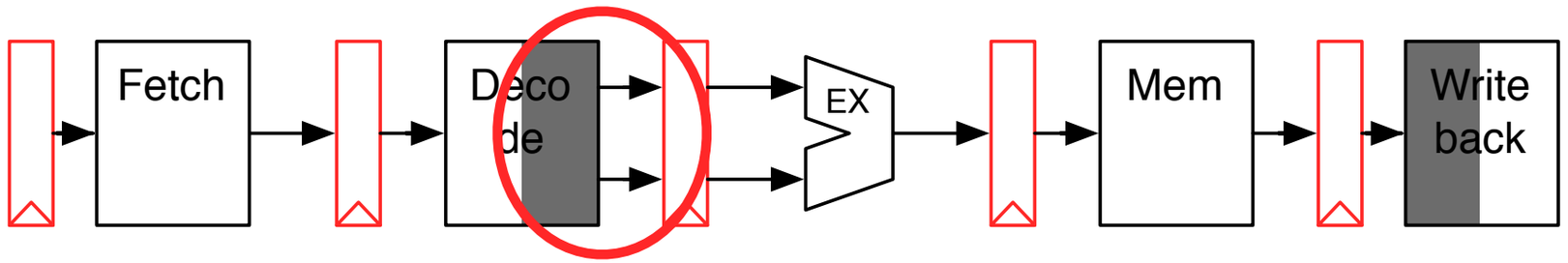
- Jump instructions
  - jr $s1 -- jump register
  - PC = $s1
- When is the value ready?



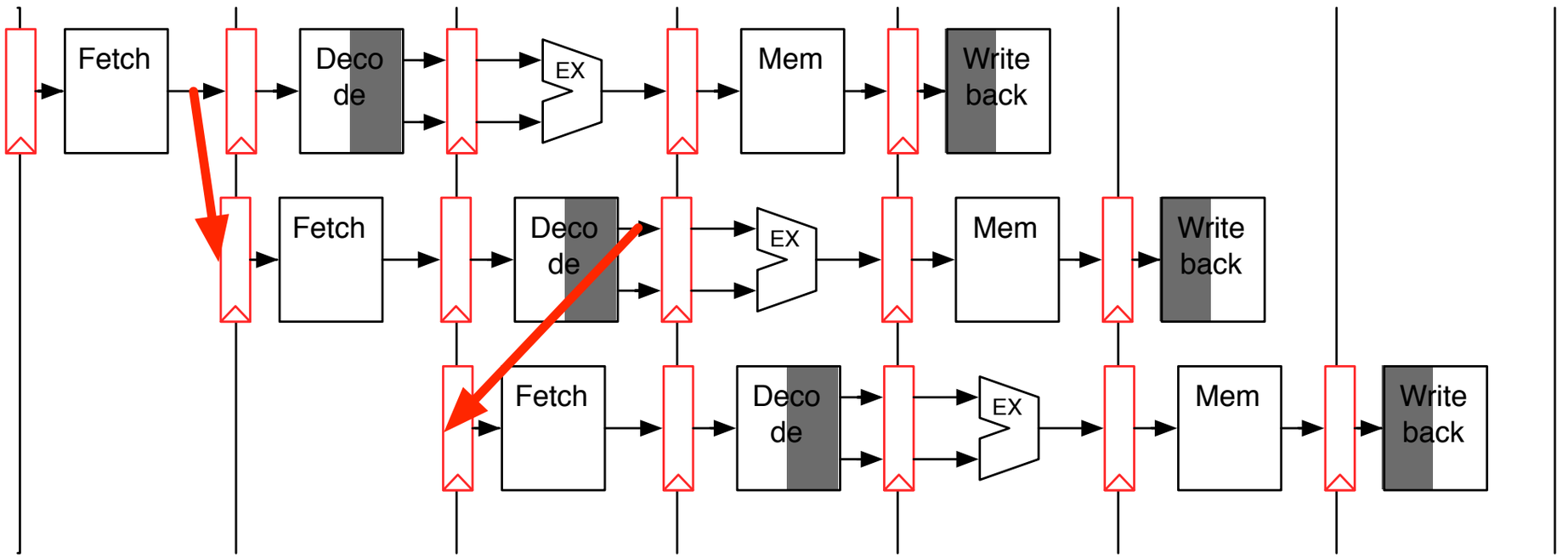


Computing the PC for Jumps

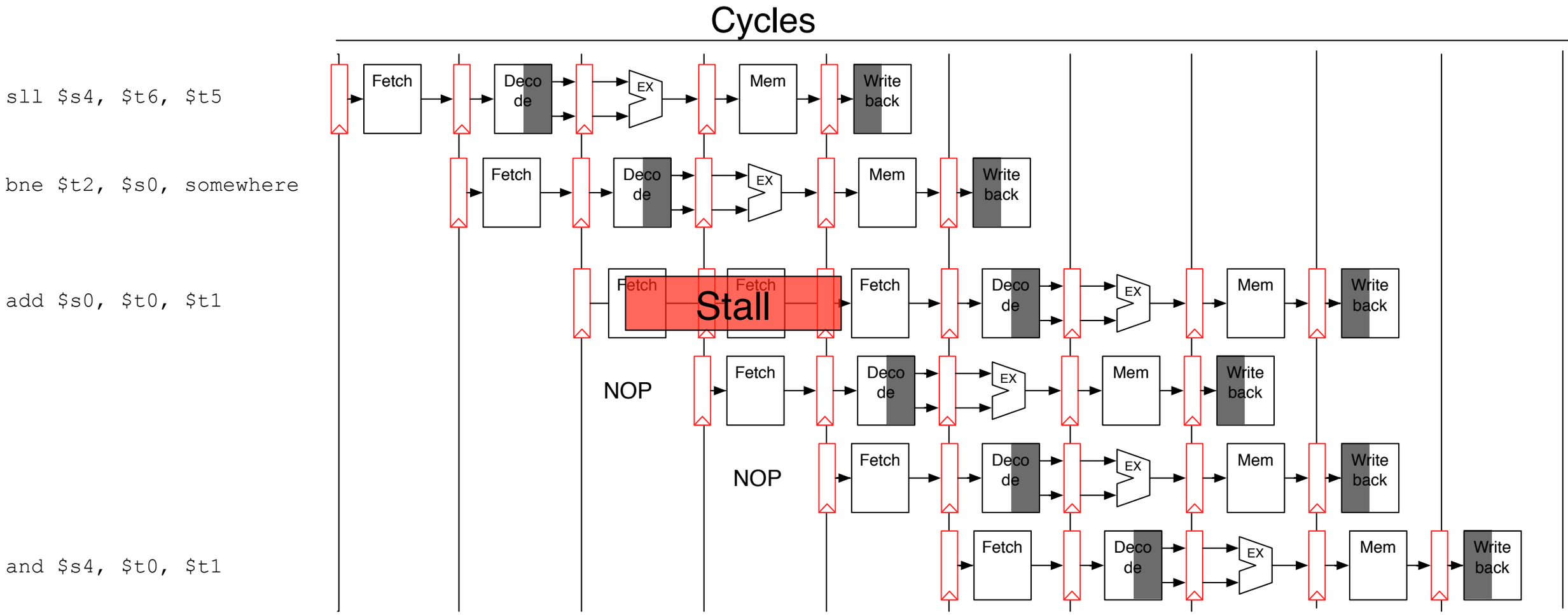
- Jump instructions
  - jr $s1 -- jump register
  - PC = $s1
- When is the value ready?




```
sll $s4, $t6, $t5

jr $s4

add $s0, $t0, $t1
```



Dealing with Branches: Option 0 -- stall



- What does this do to our CPI?

Option 1: The compiler

- Use “branch delay” slots.
- The next N instructions after a branch are *always* executed
- How big is N?
  - For jumps?
  - For branches?
- Good
  - Simple hardware
- Bad
  - N cannot change.

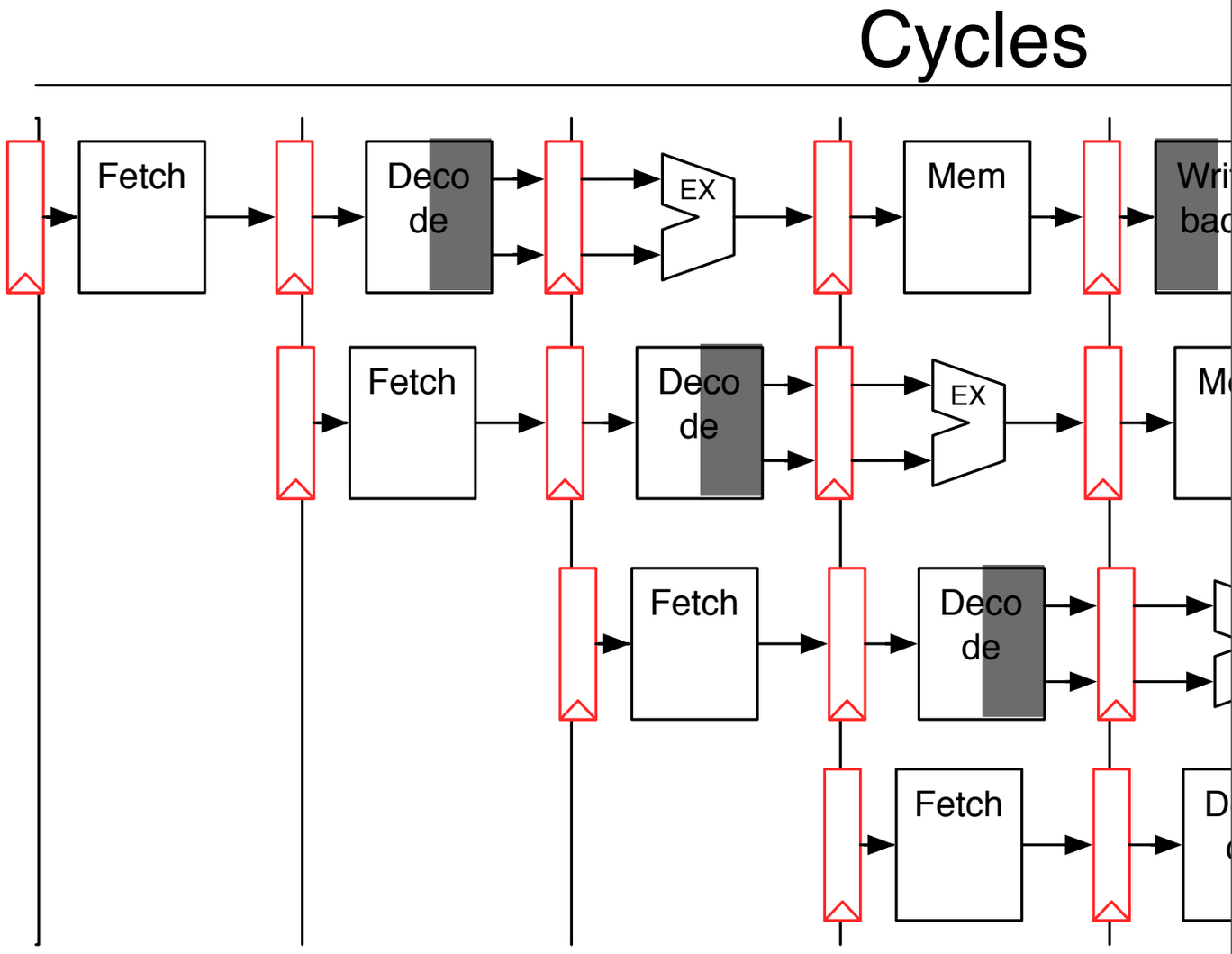
Delay slots.

Taken

```
bne $t2, $s0, somewhere
```

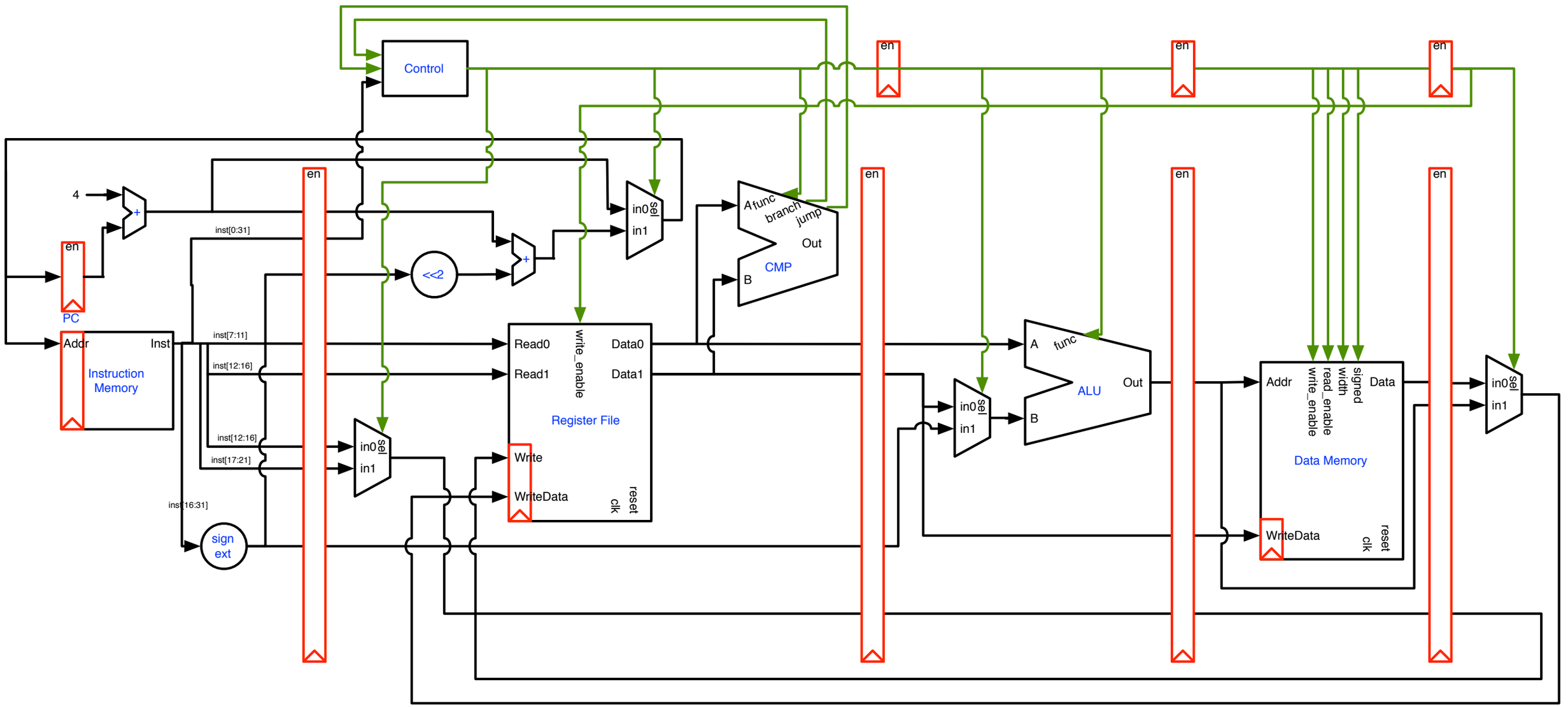
Branch Delay

```
add $t2, $s4, $t1

add $s0, $t0, $t1

...
somewhere:
sub $t2, $s0, $t3
```



But MIPS Only Has One Delay Slot!

- The second branch delay slot is expensive!
  - Filling one slot is hard. Filling two is even more so.
- Solution!: Resolve branches in decode.



For the rest of this slide deck, we will assume that MIPS has no branch delay slot.

If you have questions about whether part of the homework/test/quiz makes this assumption ask or make it clear what you assumed.

Option 2: Simple Prediction

- Can a processor tell the future?
- For non-taken branches, the new PC is ready immediately.
- Let's just assume the branch is not taken
- Also called "branch prediction" or "control speculation"
- What if we are wrong?
- Branch prediction vocabulary
  - Prediction -- a guess about whether a branch will be taken or not taken
  - Misprediction -- a prediction that turns out to be incorrect.
  - Misprediction rate -- fraction of predictions that are incorrect.

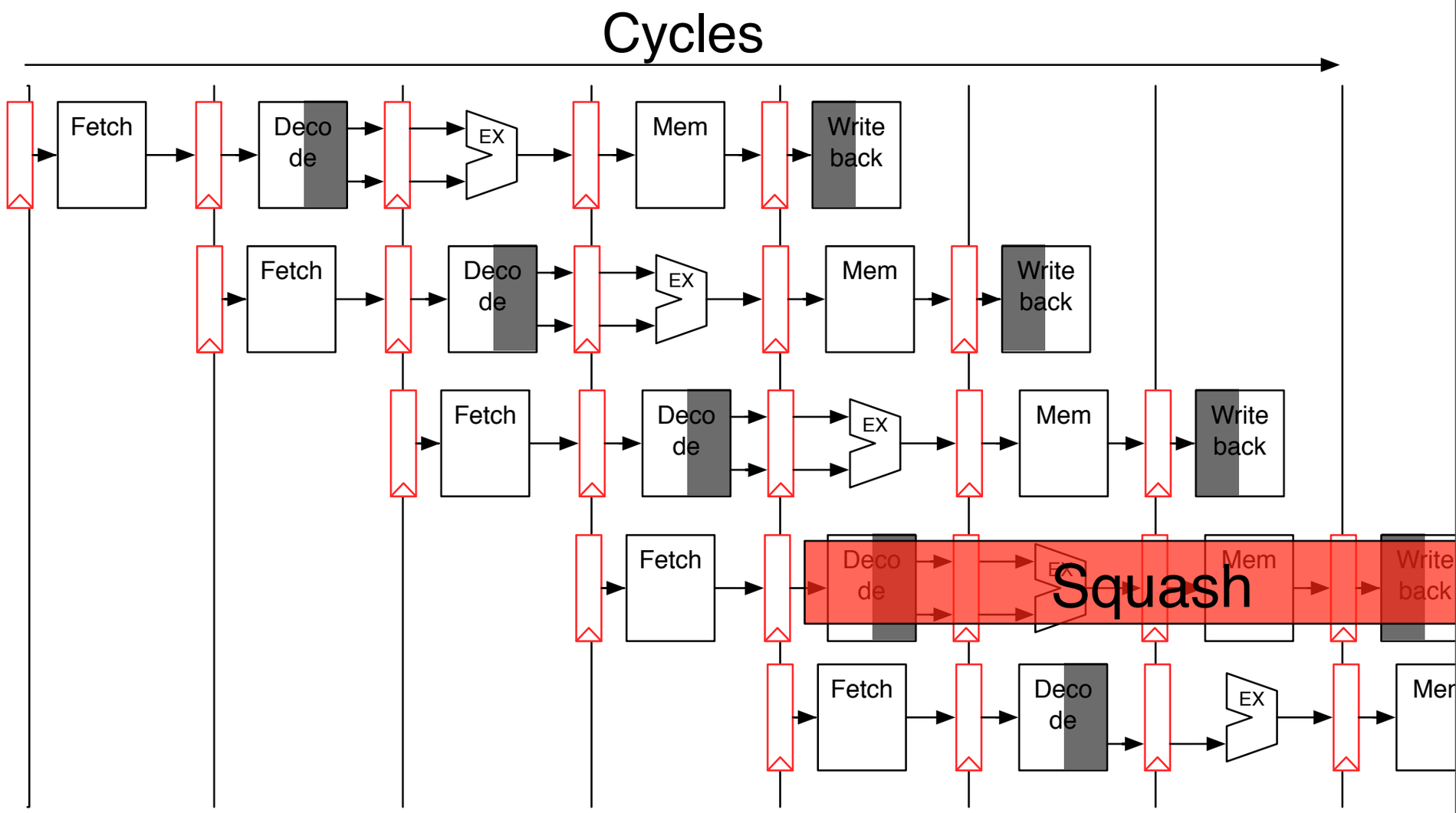
Predict Not-taken

Not-taken
```
bne $t2, $s0, foo
```

```
add $s4, $t6, $t5
```

Taken
```
bne $t2, $s4, else
```

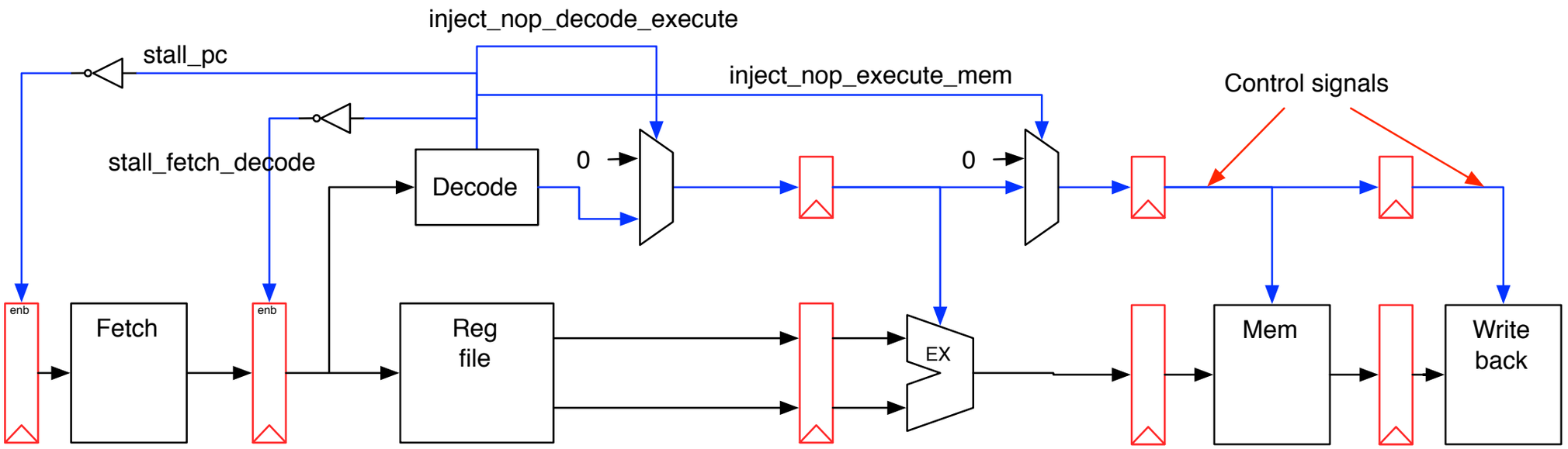
```
and $s4, $t0, $t1

...
else:
sub $t2, $s0, $t3
```



- We start the add, and then, when we discover the branch outcome, we *squash* it.
  - Also called “flushing the pipeline”
- Just like a stall, flushing one instruction increases the branch’s CPI by 1

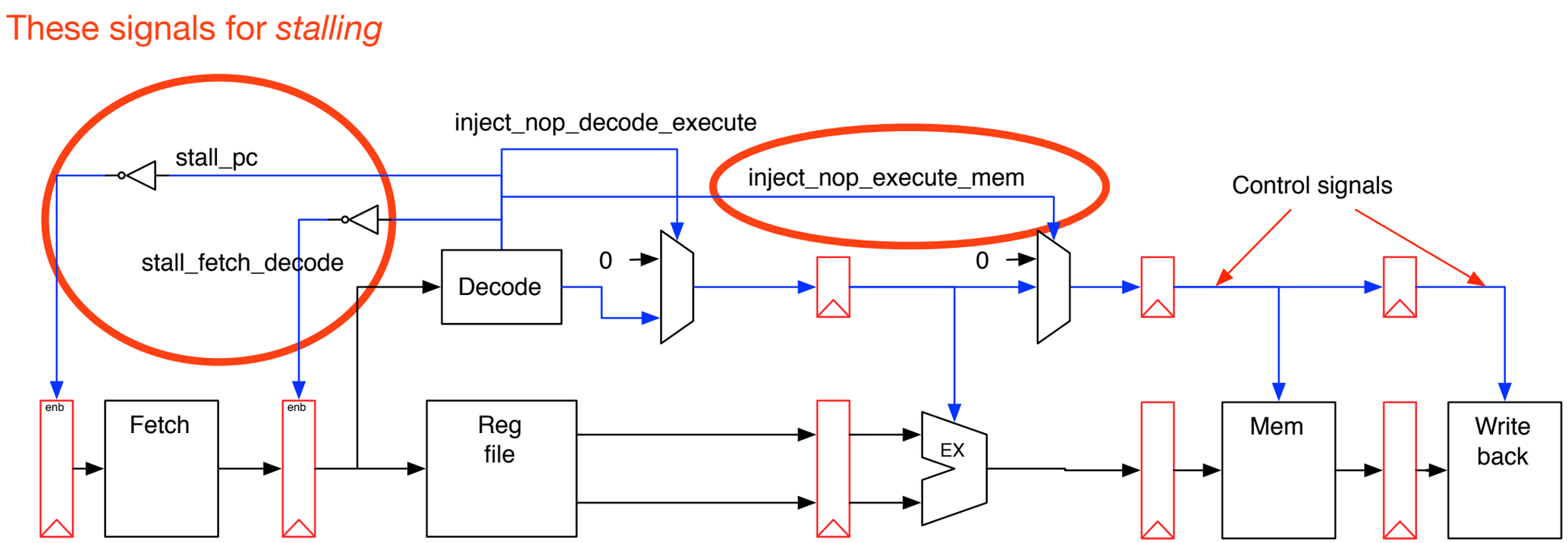
Flushing the Pipeline

- When we flush the pipe, we convert instructions into noops
  - Turn off the write enables for write back and mem stages
  - Disable branches (i.e., make sure the ALU does raise the branch signal).
- Instructions *do not stop* moving through the pipeline
- For the example on the previous slide the "inject_nop_decode_execute" signal will go high for one cycle.



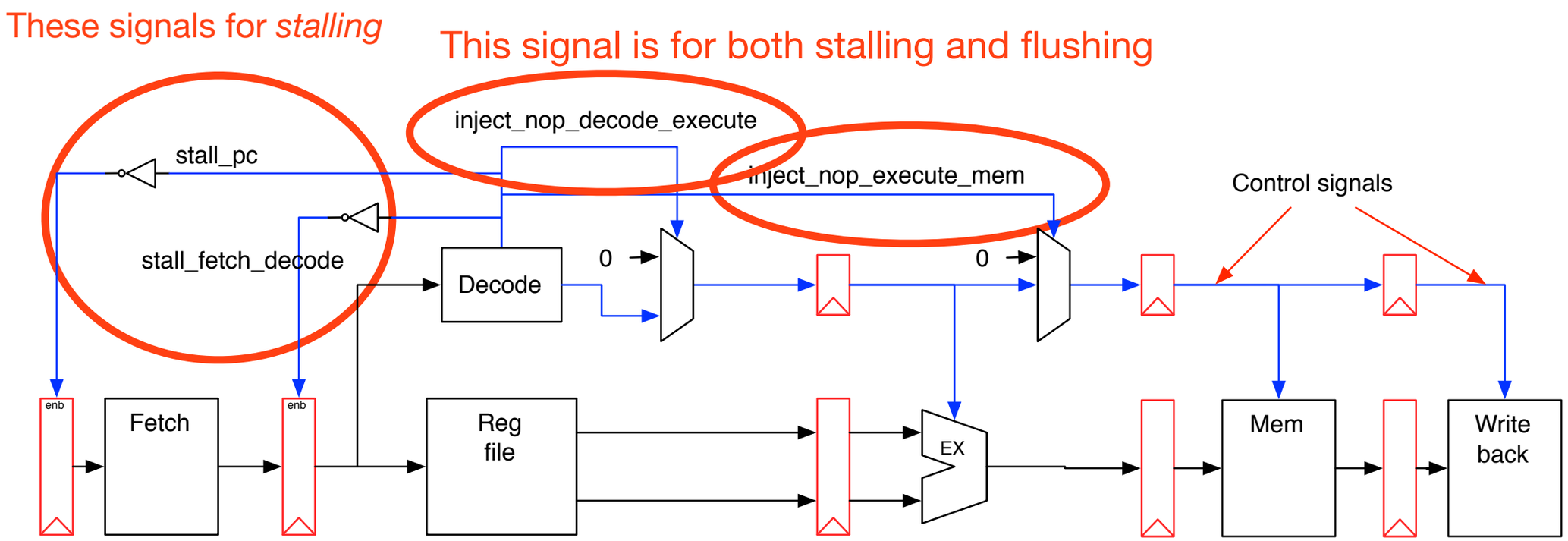
Flushing the Pipeline

- When we flush the pipe, we convert instructions into noops
  - Turn off the write enables for write back and mem stages
  - Disable branches (i.e., make sure the ALU does raise the branch signal).
- Instructions *do not stop* moving through the pipeline
- For the example on the previous slide the "inject_nop_decode_execute" signal will go high for one cycle.



Flushing the Pipeline

- When we flush the pipe, we convert instructions into noops
  - Turn off the write enables for write back and mem stages
  - Disable branches (i.e., make sure the ALU does raise the branch signal).
- Instructions *do not stop* moving through the pipeline
- For the example on the previous slide the "inject_nop_decode_execute" signal will go high for one cycle.



Simple “static” Prediction

- “static” means before run time
- Many prediction schemes are possible
- Predict taken
  - Pros?
- Predict not-taken
  - Pros?
- Backward taken/Forward not taken
  - The best of both worlds!
  - Most loops have have a backward branch at the bottom, those will predict taken
  - Others (non-loop) branches will be not-taken.

Simple “static” Prediction

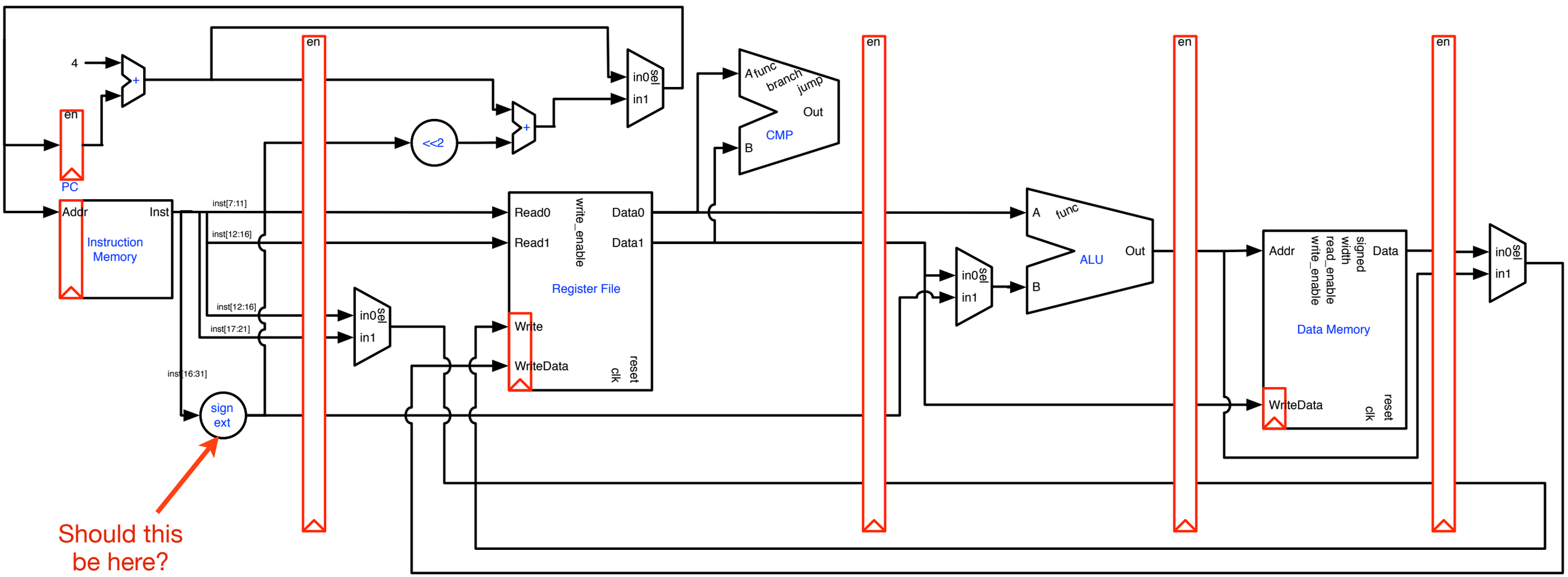
- “static” means before run time
- Many prediction schemes are possible
- Predict taken
  - Pros? Loops are commons
- Predict not-taken
  - Pros?
- Backward taken/Forward not taken
  - The best of both worlds!
  - Most loops have have a backward branch at the bottom, those will predict taken
  - Others (non-loop) branches will be not-taken.

Simple “static” Prediction

- “static” means before run time
- Many prediction schemes are possible
- Predict taken
  - Pros? Loops are commons
- Predict not-taken
  - Pros? Not all branches are for loops.
- Backward taken/Forward not taken
  - The best of both worlds!
  - Most loops have have a backward branch at the bottom, those will predict taken
  - Others (non-loop) branches will be not-taken.

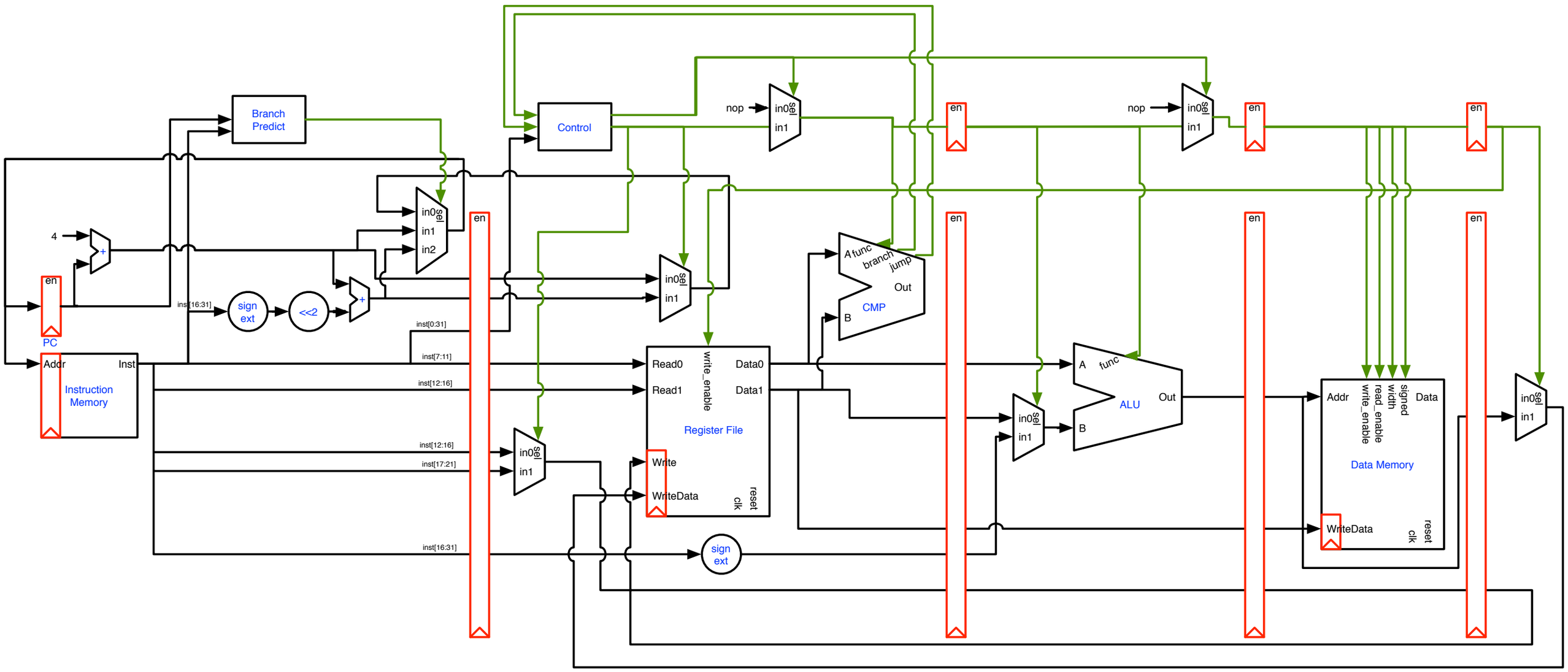
Basic Pipeline Recap

- The PC is required in Fetch
- For branches, it's not know till *decode*.



Branches only, one delay slot, simplified ISA, no control

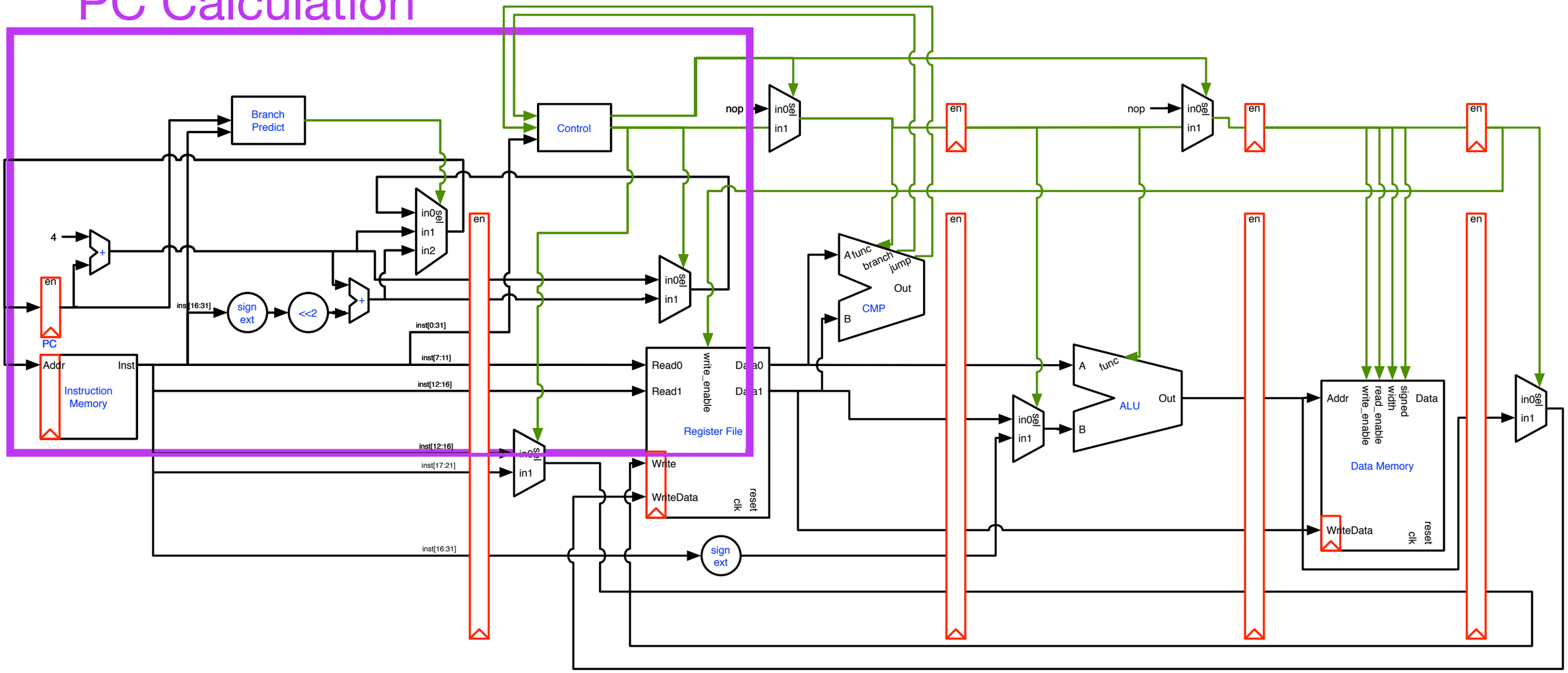
Supporting Speculation



- Predict: Compute all possible next PCs in fetch. Choose one.
  - The correct next PC is known in decode
- Flush as needed: Replace “wrong path” instructions with no-ops.

Supporting Speculation

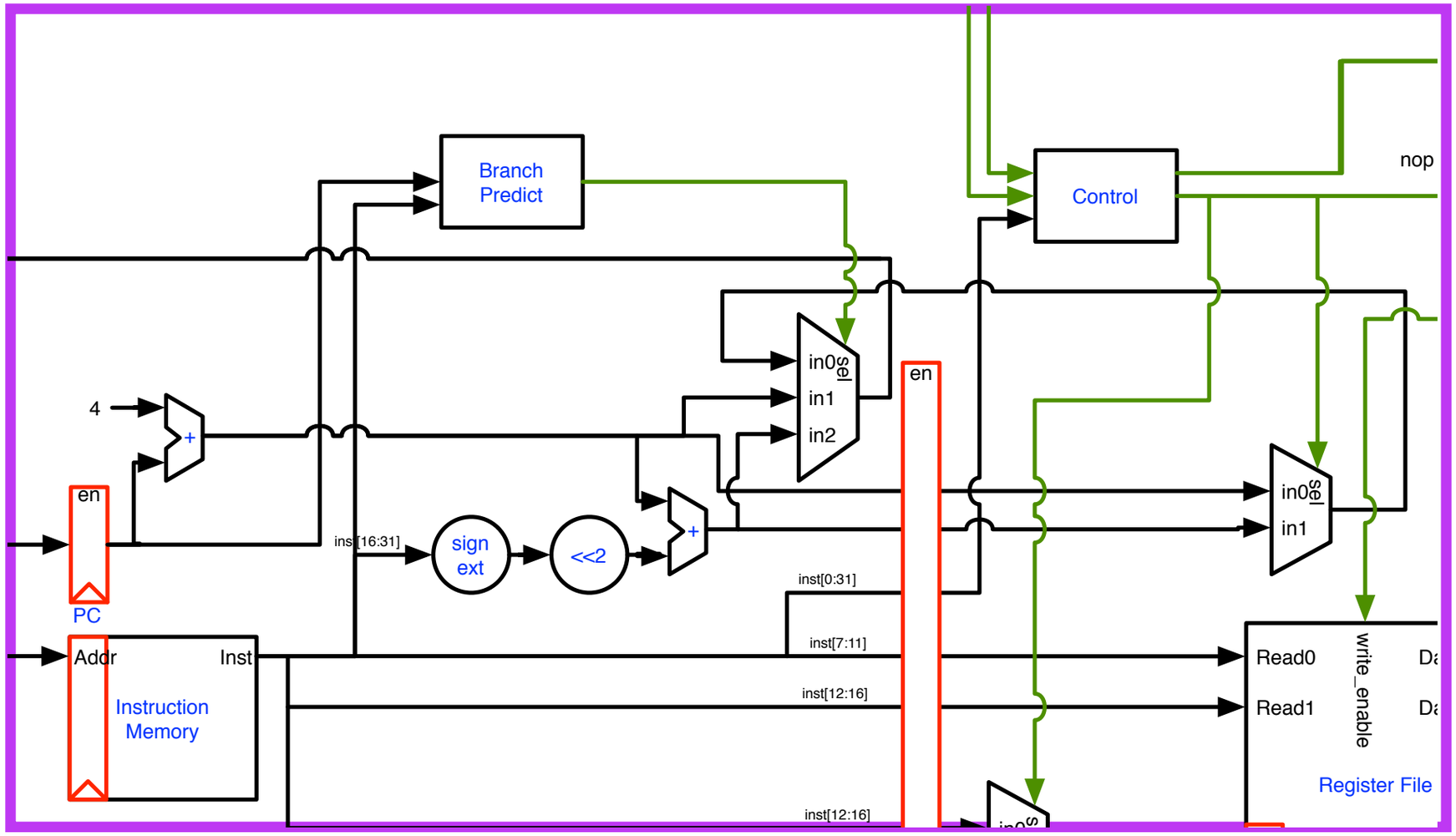
PC Calculation



- Predict: Compute all possible next PCs in fetch. Choose one.
  - The correct next PC is known in decode
- Flush as needed: Replace “wrong path” instructions with no-ops.

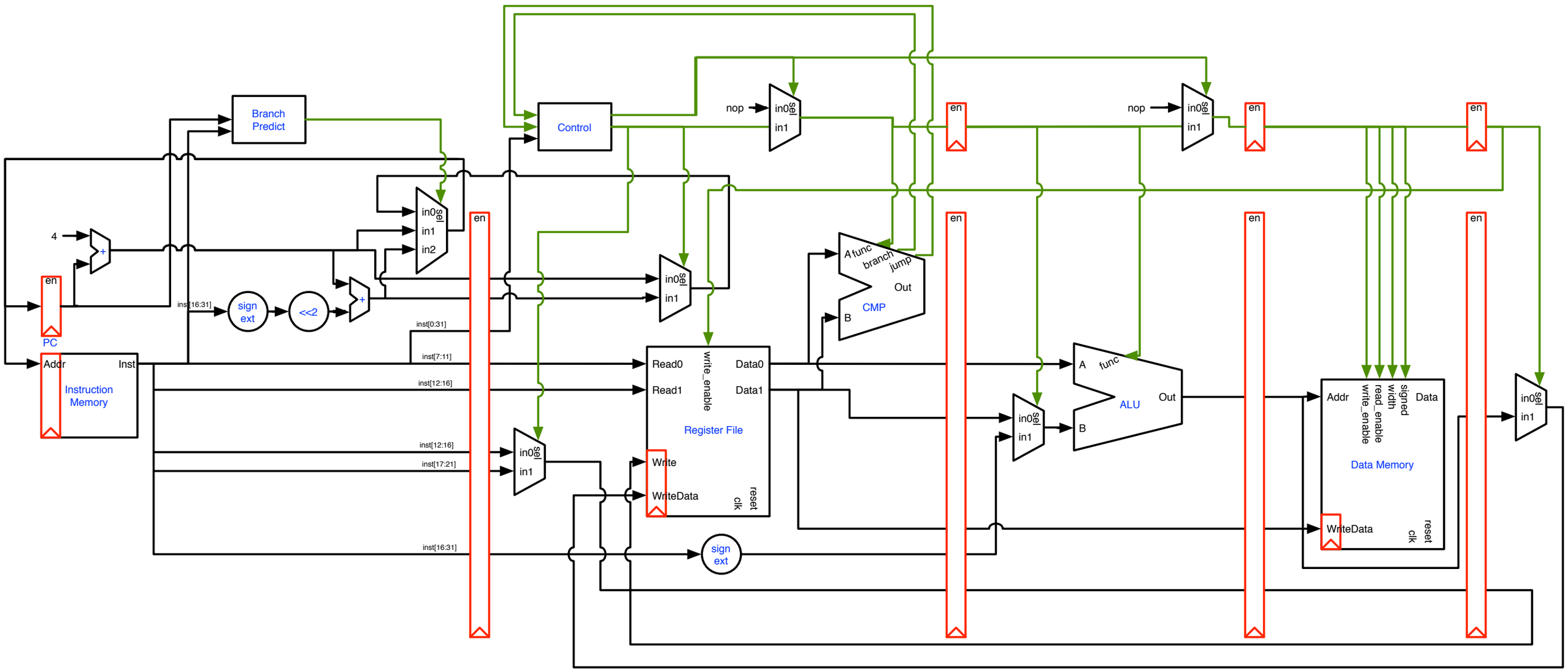
Supporting Speculation

PC Calculation



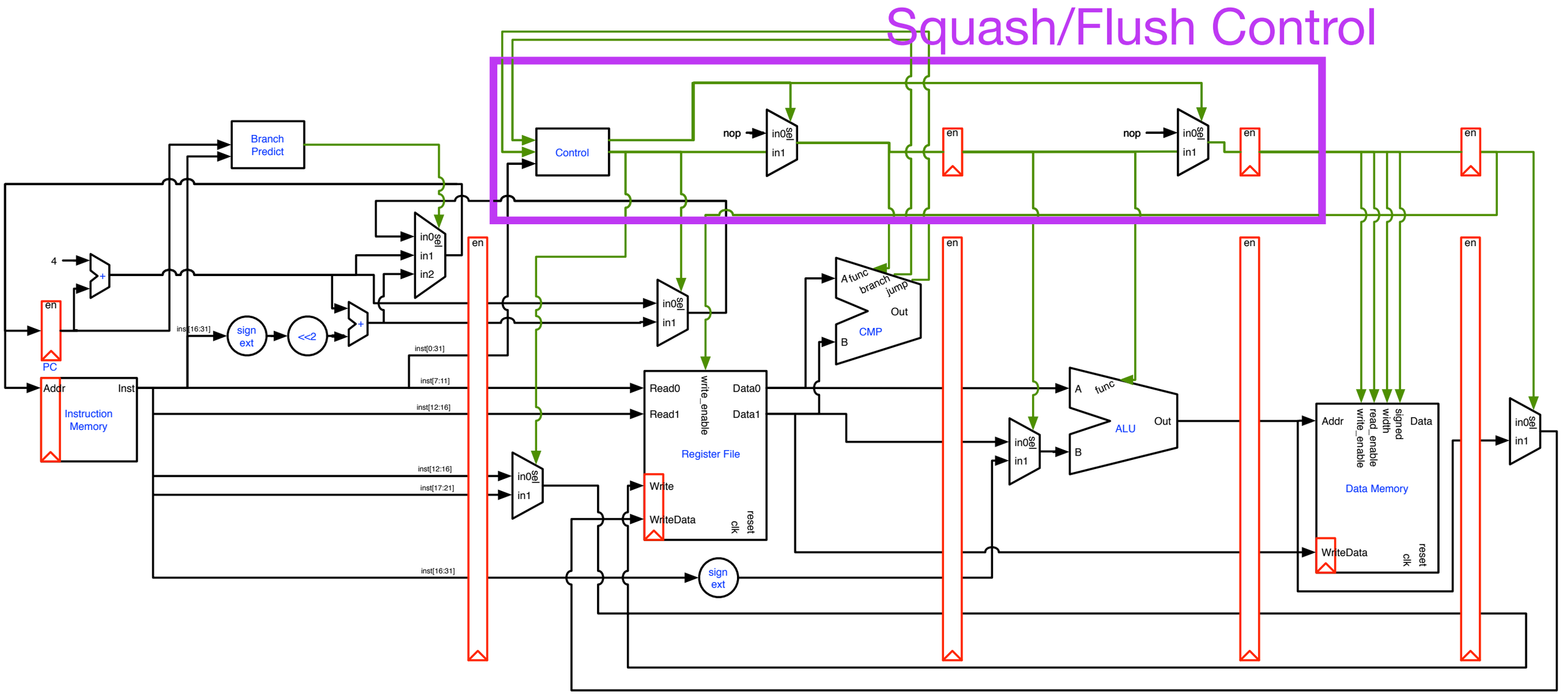
- Predict: Compute all possible next PCs in fetch. Choose one.
  - The correct next PC is known in decode
- Flush as needed: Replace “wrong path” instructions with no-ops.

Supporting Speculation



- Predict: Compute all possible next PCs in fetch. Choose one.
  - The correct next PC is known in decode
- Flush as needed: Replace “wrong path” instructions with no-ops.

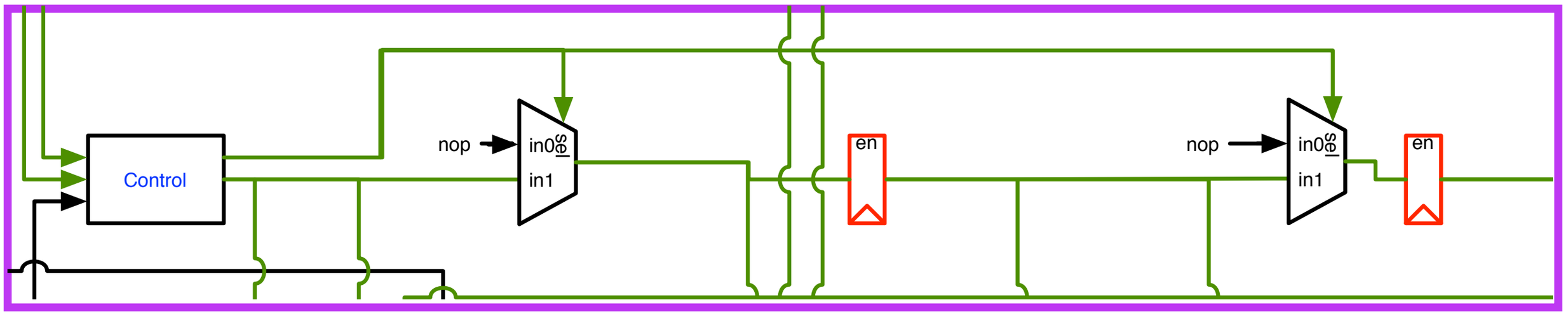
Supporting Speculation



- Predict: Compute all possible next PCs in fetch. Choose one.
  - The correct next PC is known in decode
- Flush as needed: Replace “wrong path” instructions with no-ops.

Supporting Speculation

Squash/Flush Contro



- Predict: Compute all possible next PCs in fetch. Choose one.
  - The correct next PC is known in decode
- Flush as needed: Replace “wrong path” instructions with no-ops.

Implementing Backward taken/forward not taken (BTFNT)

- A new “branch predictor” module determines what guess we are going to make.
- The BTFNT branch predictor has one input
  - The sign of the offset -- to make the prediction
  - The branch signal from the comparator -- to check if the prediction was correct.
- And two output
  - The PC mux selector
    - Steers execution in the predicted direction
    - Re-directs execution when the branch resolves.
  - A mis-predict signal that causes control to flush the pipe.

Performance Impact (ex 1)

- ET = I * CPI * CT

- BTFTN is has a misprediction rate of 20%.
- Branches are 20% of instructions
- Changing the front end increases the cycle time by 10%
- What is the speedup BTFNT compared to just stalling on every branch?

Performance Impact (ex 1)

- ET = I * CPI * CT

- Back taken, forward not taken is 80% accurate
- Branches are 20% of instructions
- Changing the front end increases the cycle time by 10%
- What is the speedup Bt/Fnt compared to just stalling on every branch?
- Btfnt
  - CPI = 0.2*0.2*(1 + 1) + (1-.2*.2)*1 = 1.04
  - CT = 1.1
  - IC = IC
  - ET = 1.144
- Stall
  - CPI = .2*2 + .8*1 = 1.2
  - CT = 1
  - IC = IC
  - ET = 1.2
- Speed up = 1.2/1.144 = 1.05

The Branch Delay Penalty

- The number of cycle between fetch and branch resolution is called the “branch delay penalty”
  - It is the number of instruction that get flushed on a misprediction.
  - It is the number of extra cycles the branch gets charged (i.e., the CPI for mispredicted branches goes up by the penalty for)

Performance Impact (ex 2)

- ET = I * CPI * CT
- Our current design resolves branches in decode, so the branch delay penalty is 1 cycle.
- If removing the comparator from decode (and resolving branches in execute) would reduce cycle time by 20%, would it help or hurt performance?
  - Mis predict rate = 20%
  - Branches are 20% of instructions
- Resolve in Decode
  - CPI = 0.2*0.2*(1 + 1) + (1-.2*.2)*1 = 1.04
  - CT = 1
  - IC = IC
  - ET = 1.04
- Resolve in execute
  - CPI = 0.2*0.2*(1 + 2) + (1-.2*.2)*1 = 1.08
  - CT = 0.8
  - IC = IC
  - ET = 0.864
- Speedup = 1.2

Performance Impact (ex 2)

- ET = I * CPI * CT
- Our current design resolves branches in decode, so the branch delay penalty is 1 cycle.
- If removing the comparator from decode (and resolving branches in execute) would reduce cycle time by 20%, would it help or hurt performance?
  - Mis predict rate = 20%
  - Branches are 20% of instructions

The Importance of Pipeline depth

- There are two important parameters of the pipeline that determine the impact of branches on performance
  - Branch decode time -- how many cycles does it take to identify a branch (in our case, this is less than 1)
  - Branch resolution time -- cycles until the real branch outcome is known (in our case, this is 2 cycles)

Pentium 4 pipeline

- Branches take 19 cycles to resolve
- Identifying a branch takes 4 cycles.
- Stalling is not an option.
- 80% branch prediction accuracy is also not an option.
- Not quite as bad now, but BP is still very important.

Performance Impact (ex 1)

- ET = I * CPI * CT

- Back taken, forward not taken is 80% accurate
- Branches are 20% of instructions
- Changing the front end increases the cycle time by 10%
- What is the speedup Bt/Fnt compared to just stalling on every branch?
- Btfnt
  - CPI = 0.2*0.2*(1 + 1) + (1-.2*.2)*1 = 1.04
  - CT = 1.144
  - IC = IC
  - ET = 1.144
- Stall
  - CPI = .2*2 + .8*1 = 1.2
  - CT = 1
  - IC = IC
  - ET = 1.2
- Speed up = 1.2/1.144 = 1.05

What if this were 20 instead of 1?

Performance Impact (ex 1)

- ET = I * CPI * CT

- Back taken, forward not taken is 80% accurate
- Branches are 20% of instructions
- Changing the front end increases the cycle time by 10%
- What is the speedup Bt/Fnt compared to just stalling on every branch?
- Btfnt
  - CPI = 0.2*0.2*(1 + 1) + (1-.2*.2)*1 = 1.04
  - CT = 1.144
  - IC = IC
  - ET = 1.144
- Stall
  - CPI = .2*2 + .8*1 = 1.2
  - CT = 1
  - IC = IC
  - ET = 1.2
- Speed up = 1.2/1.144 = 1.05

What if this were 20 instead of 1?

Branches are relatively infrequent (~20% of instructions), but Amdahl's Law tells that we can't completely ignore this uncommon case.

Dynamic Branch Prediction

- Long pipes demand higher accuracy than static schemes can deliver.
- Instead of making the the guess once (i.e. statically), make it every time we see the branch.
- Many ways to predict dynamically
  - We will focus on predicting future behavior based on past behavior

Predictable control

- Use previous branch behavior to predict future branch behavior.
- When is branch behavior predictable?

Predictable control

- Use previous branch behavior to predict future branch behavior.
- When is branch behavior predictable?
  - Loops -- for(i = 0; i < 10; i++) {} 9 taken branches, 1 not-taken branch. All 10 are pretty predictable.
  - Run-time constants
    - Foo(int v,) { for (i = 0; i < 1000; i++) {if (v) {...}}}.
    - The branch is always taken or not taken.
  - Corollated control
    - a = 10; b = <something usually larger than a >
    - if (a > 10) {}
    - if (b > 10) {}
  - Function calls
    - LibraryFunction() -- Converts to a jr (jump register) instruction, but it's always the same.
    - BaseClass * t;   // t is usually a of sub class, SubClass
    - t->SomeVirtualFunction() // will usually call the same function

Dynamic Predictor 1: The Simplest Thing

- Predict that this branch will go the same way as the previous branch did.
- Pros?
- Cons?

Dynamic Predictor 1: The Simplest Thing

- Predict that this branch will go the same way as the previous branch did.
- Pros?

Dead simple. Keep a bit in the fetch stage. Works ok for simple loops. The compiler might be able to arrange things to make it work better.

- Cons?

Dynamic Predictor 1: The Simplest Thing

- Predict that this branch will go the same way as the previous branch did.
- Pros?

Dead simple. Keep a bit in the fetch stage. Works ok for simple loops. The compiler might be able to arrange things to make it work better.

- Cons?

An unpredictable branch in a loop will mess everything up. It can't tell the difference between branches

Dynamic Prediction 2: A table of bits

- Give each branch it's own bit in a table
  - Look up the prediction bit for the branch
  - How big does the table need to be?

- Pros:

- Cons:

Dynamic Prediction 2: A table of bits

- Give each branch it's own bit in a table
  - Look up the prediction bit for the branch
  - How big does the table need to be?

- Pros:

  It can differentiate between branches. Bad behavior by one won't mess up others.... mostly.

- Cons:

Dynamic Prediction 2: A table of bits

- Give each branch it's own bit in a table
  - Look up the prediction bit for the branch
  - How big does the table need to be?

Infinite! Bigger is better, but don't mess with the cycle time. Index into it using the low order bits of the PC

- Pros:

It can differentiate between branches. Bad behavior by one won't mess up others.... mostly.

- Cons:

Dynamic Prediction 2: A table of bits

- Give each branch it's own bit in a table
  - Look up the prediction bit for the branch
  - How big does the table need to be?

Infinite! Bigger is better, but don't mess with the cycle time. Index into it using the low order bits of the PC

- Pros:

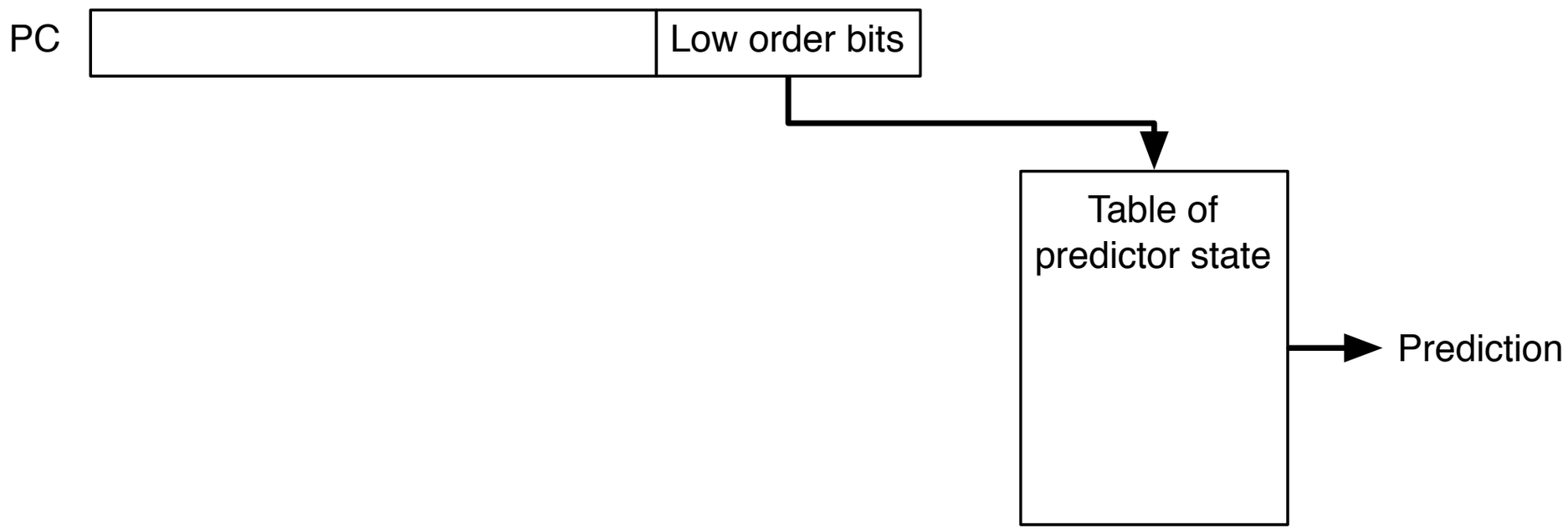
It can differentiate between branches. Bad behavior by one won't mess up others.... mostly.

- Cons:

Accuracy is still not great.

Branch Prediction Trick #1

- Associating prediction state with a particular branch.
- We would like to keep separate prediction state for every *static* branch.
  - In practice this is not possible, since there are a potentially unbounded number of branches
- Instead, we use a heuristic to associate prediction state with a branch
  - The simplest heuristic is to use the low-order bits of the PC to select the prediction state.



Dynamic Prediction 2: A table of bits

```
while (1) {
  for(j = 0; j < 4; j++) { // branch at
                           address 0x100A

  }
}
```

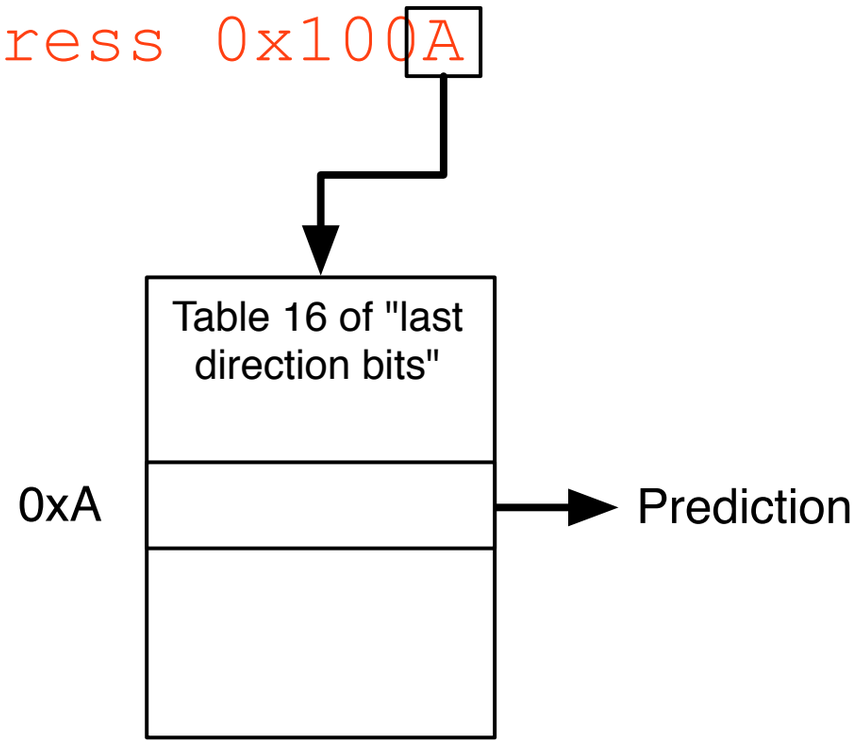
iteration	Actual	prediction	new prediction
1	taken	not taken	taken
2	taken	taken	taken
3	taken	taken	taken
4	not taken	taken	not taken
1	taken	not taken	take
2	taken	taken	taken
3	taken	taken	taken

- What's the accuracy for the branch?

Dynamic Prediction 2: A table of bits

```
while (1) {
  for(j = 0; j < 4; j++) { // branch at
                           address 0x100A

  }
}
```

iteration	Actual	prediction	new prediction
1	taken	not taken	taken
2	taken	taken	taken
3	taken	taken	taken
4	not taken	taken	not taken
1	taken	not taken	take
2	taken	taken	taken
3	taken	taken	taken

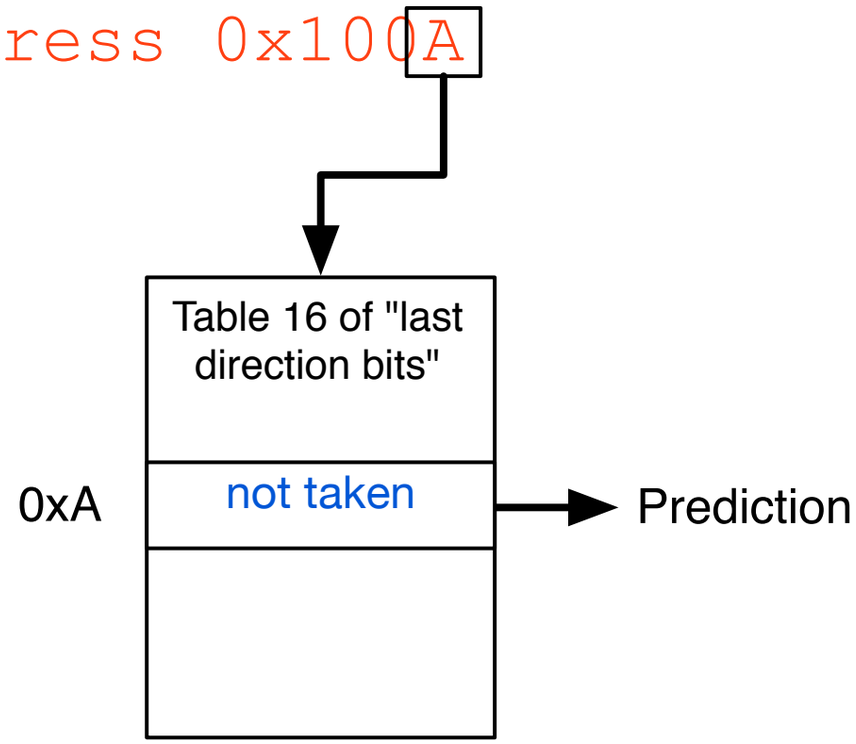


- What's the accuracy for the branch?

Dynamic Prediction 2: A table of bits

```
while (1) {
  for(j = 0; j < 4; j++) { // branch at
                           address 0x100A

  }
}
```

iteration	Actual	prediction	new prediction
1	taken	not taken	taken
2	taken	taken	taken
3	taken	taken	taken
4	not taken	taken	not taken
1	taken	not taken	take
2	taken	taken	taken
3	taken	taken	taken

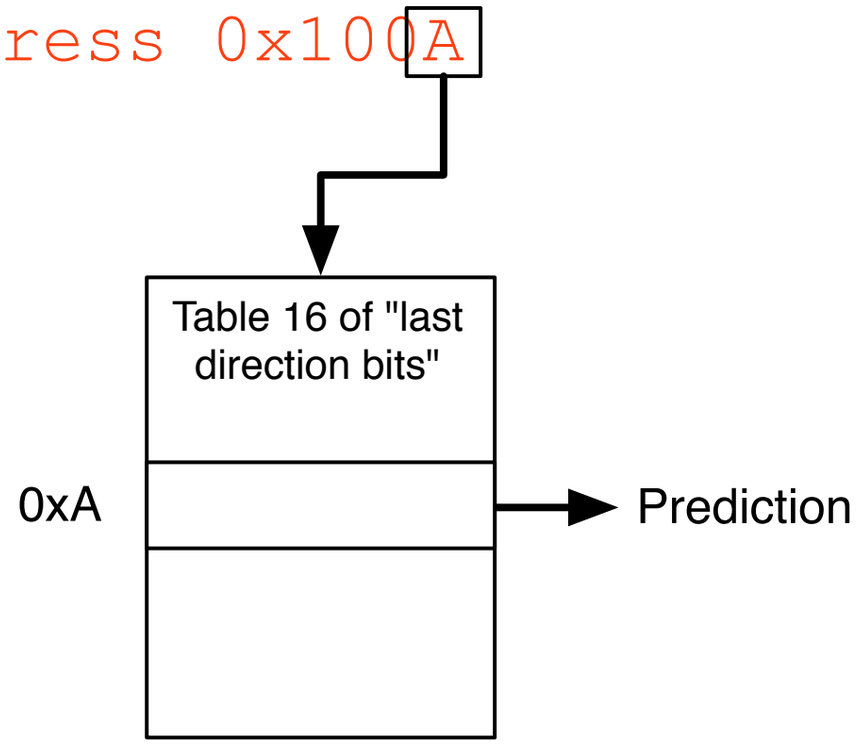


- What's the accuracy for the branch?

Dynamic Prediction 2: A table of bits

```
while (1) {
  for(j = 0; j < 4; j++) { // branch at
                           address 0x100A

  }
}
```

iteration	Actual	prediction	new prediction
1	taken	not taken	taken
2	taken	taken	taken
3	taken	taken	taken
4	not taken	taken	not taken
1	taken	not taken	take
2	taken	taken	taken
3	taken	taken	taken

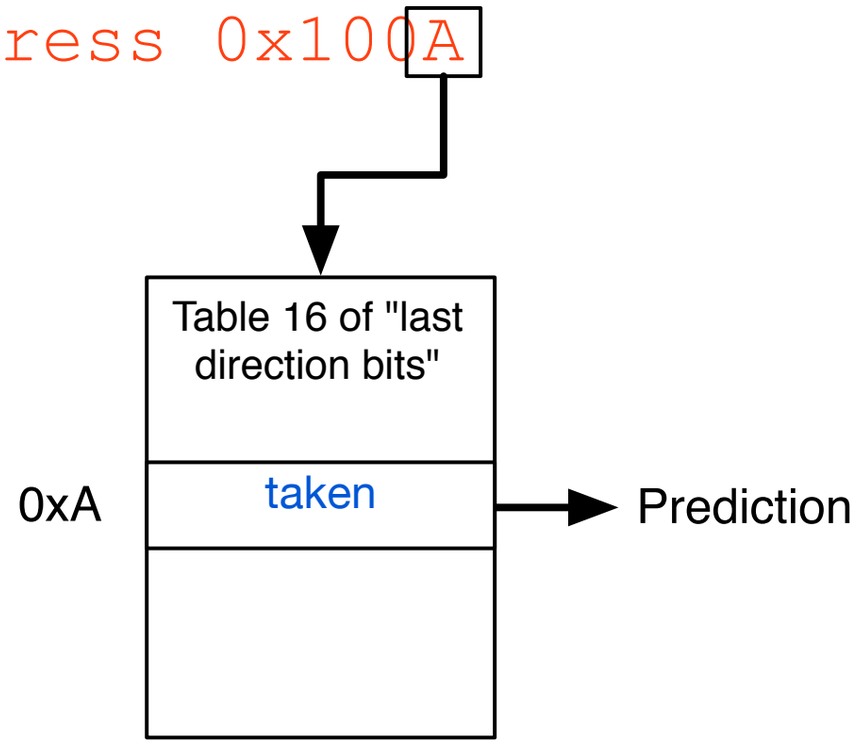


- What's the accuracy for the branch?

Dynamic Prediction 2: A table of bits

```
while (1) {
  for(j = 0; j < 4; j++) { // branch at
                           address 0x100A

  }
}
```

iteration	Actual	prediction	new prediction
1	taken	not taken	taken
2	taken	taken	taken
3	taken	taken	taken
4	not taken	taken	not taken
1	taken	not taken	take
2	taken	taken	taken
3	taken	taken	taken

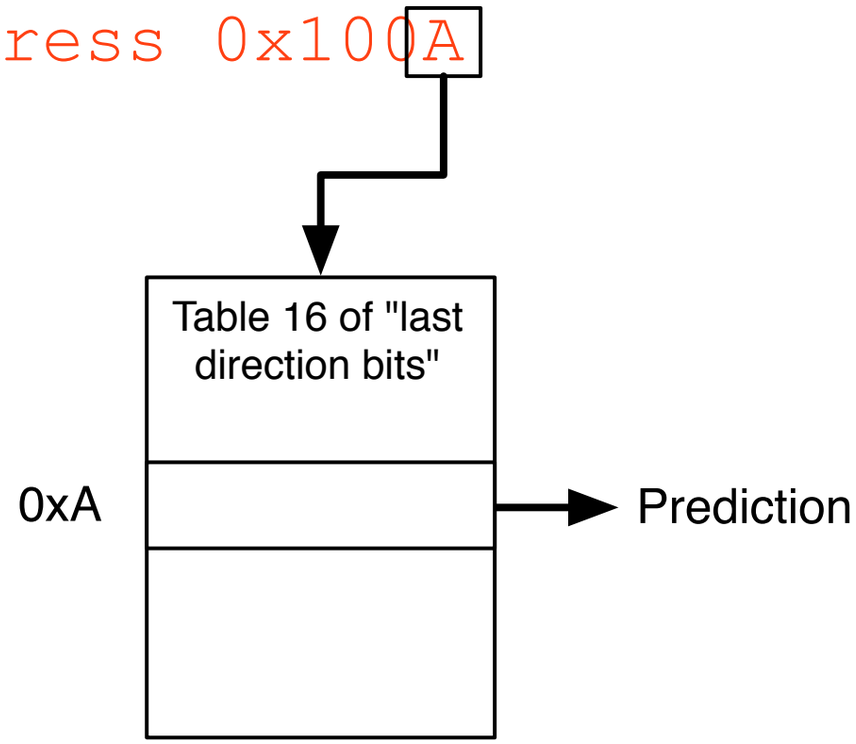


- What's the accuracy for the branch?

Dynamic Prediction 2: A table of bits

```
while (1) {
  for(j = 0; j < 4; j++) { // branch at
                           address 0x100A

  }
}
```

iteration	Actual	prediction	new prediction
1	taken	not taken	taken
2	taken	taken	taken
3	taken	taken	taken
4	not taken	taken	not taken
1	taken	not taken	take
2	taken	taken	taken
3	taken	taken	taken

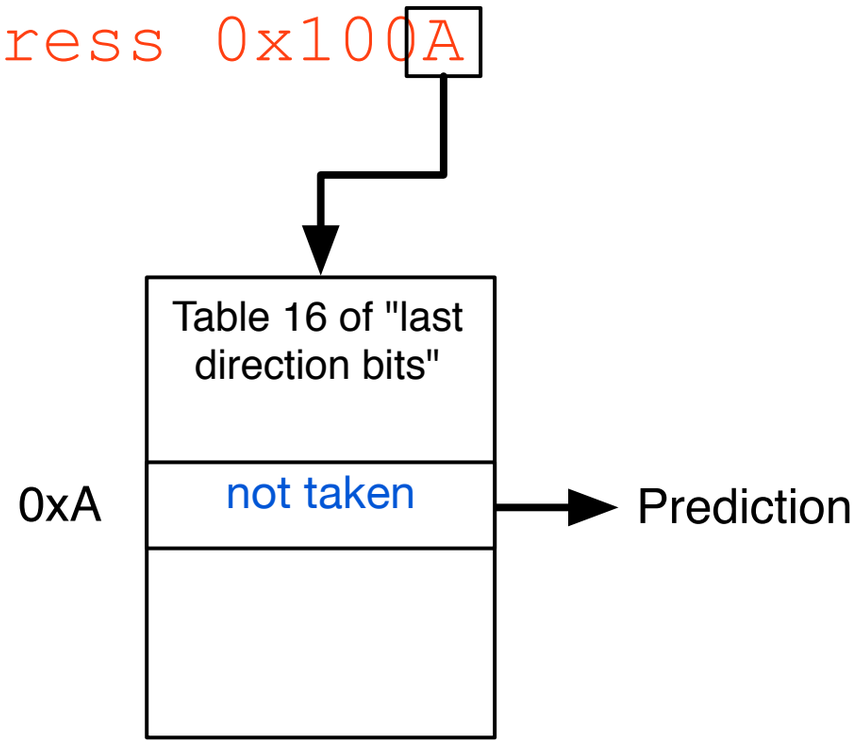


- What's the accuracy for the branch?

Dynamic Prediction 2: A table of bits

```
while (1) {
  for(j = 0; j < 4; j++) { // branch at
                           address 0x100A

  }
}
```

iteration	Actual	prediction	new prediction
1	taken	not taken	taken
2	taken	taken	taken
3	taken	taken	taken
4	not taken	taken	not taken
1	taken	not taken	take
2	taken	taken	taken
3	taken	taken	taken

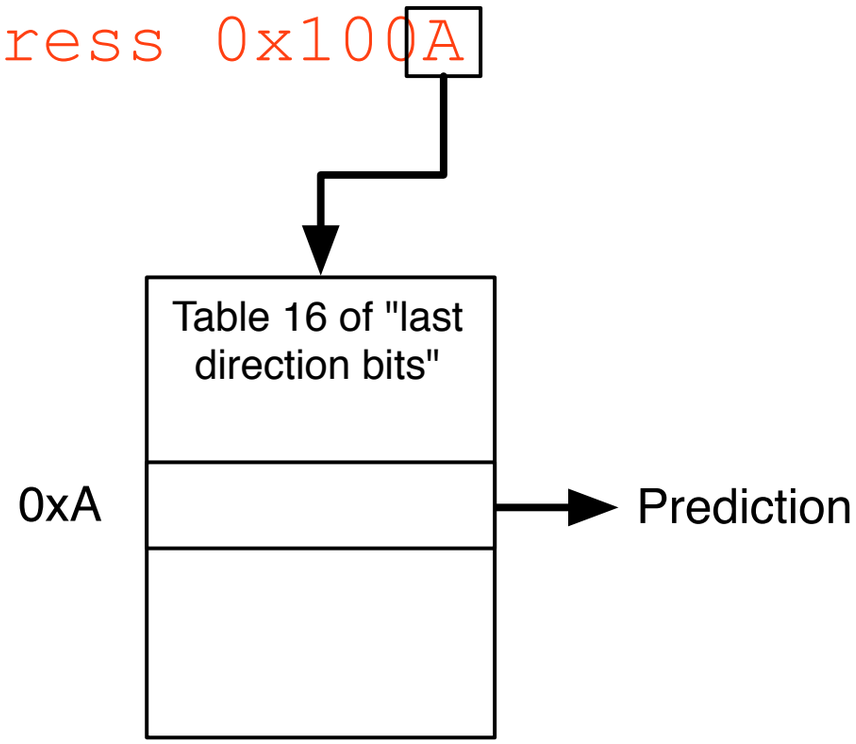


- What's the accuracy for the branch?

Dynamic Prediction 2: A table of bits

```
while (1) {
  for(j = 0; j < 4; j++) { // branch at
                           address 0x100A

  }
}
```

iteration	Actual	prediction	new prediction
1	taken	not taken	taken
2	taken	taken	taken
3	taken	taken	taken
4	not taken	taken	not taken
1	taken	not taken	take
2	taken	taken	taken
3	taken	taken	taken

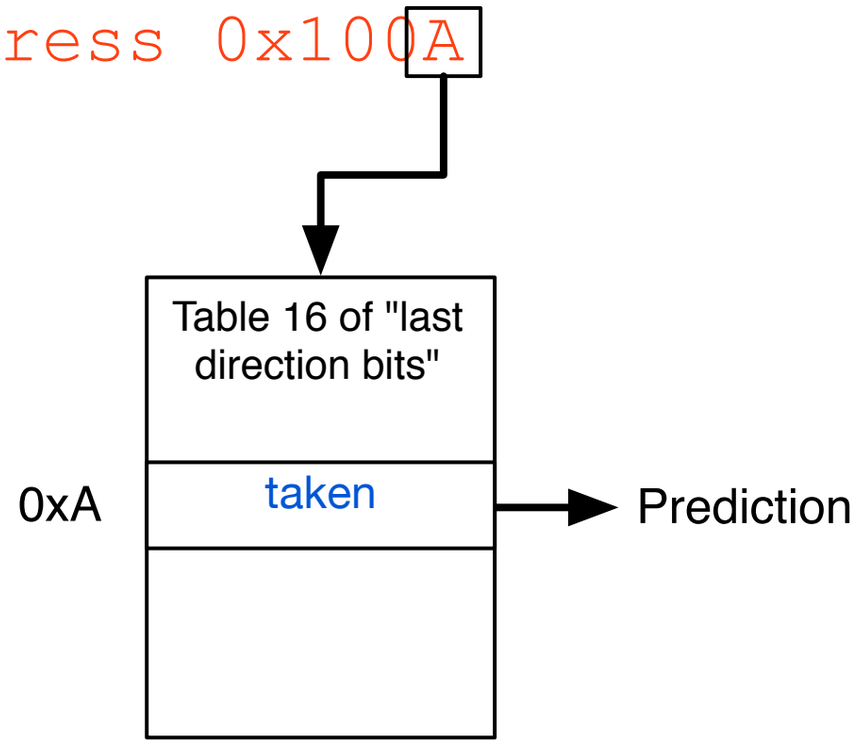


- What's the accuracy for the branch?

Dynamic Prediction 2: A table of bits

```
while (1) {
  for(j = 0; j < 4; j++) { // branch at
                           address 0x100A

  }
}
```

iteration	Actual	prediction	new prediction
1	taken	not taken	taken
2	taken	taken	taken
3	taken	taken	taken
4	not taken	taken	not taken
1	taken	not taken	take
2	taken	taken	taken
3	taken	taken	taken

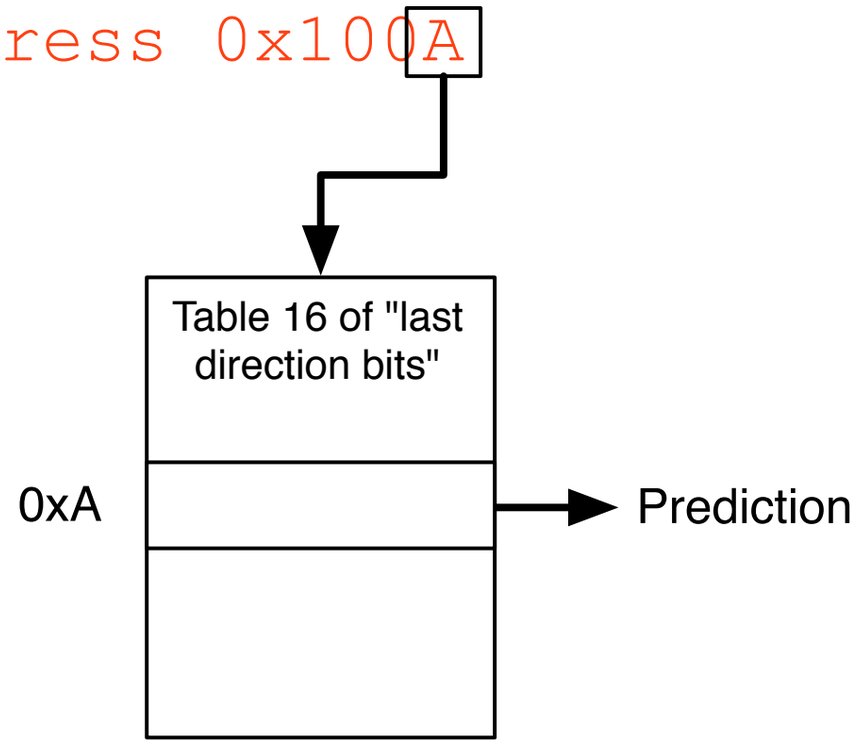


- What's the accuracy for the branch?

Dynamic Prediction 2: A table of bits

```
while (1) {
  for(j = 0; j < 4; j++) { // branch at
                           address 0x100A

  }
}
```

iteration	Actual	prediction	new prediction
1	taken	not taken	taken
2	taken	taken	taken
3	taken	taken	taken
4	not taken	taken	not taken
1	taken	not taken	take
2	taken	taken	taken
3	taken	taken	taken

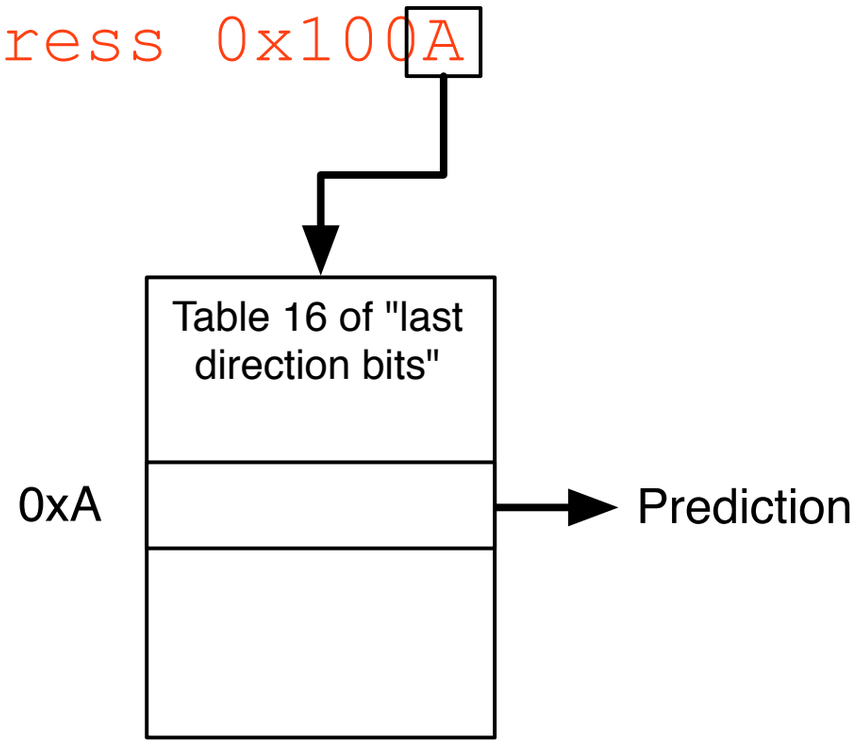


- What's the accuracy for the branch?

Dynamic Prediction 2: A table of bits

```
while (1) {
  for(j = 0; j < 4; j++) { // branch at
                           address 0x100A

  }
}
```

iteration	Actual	prediction	new prediction
1	taken	not taken	taken
2	taken	taken	taken
3	taken	taken	taken
4	not taken	taken	not taken
1	taken	not taken	take
2	taken	taken	taken
3	taken	taken	taken



- What's the accuracy for the branch?

50% or 2 per loop

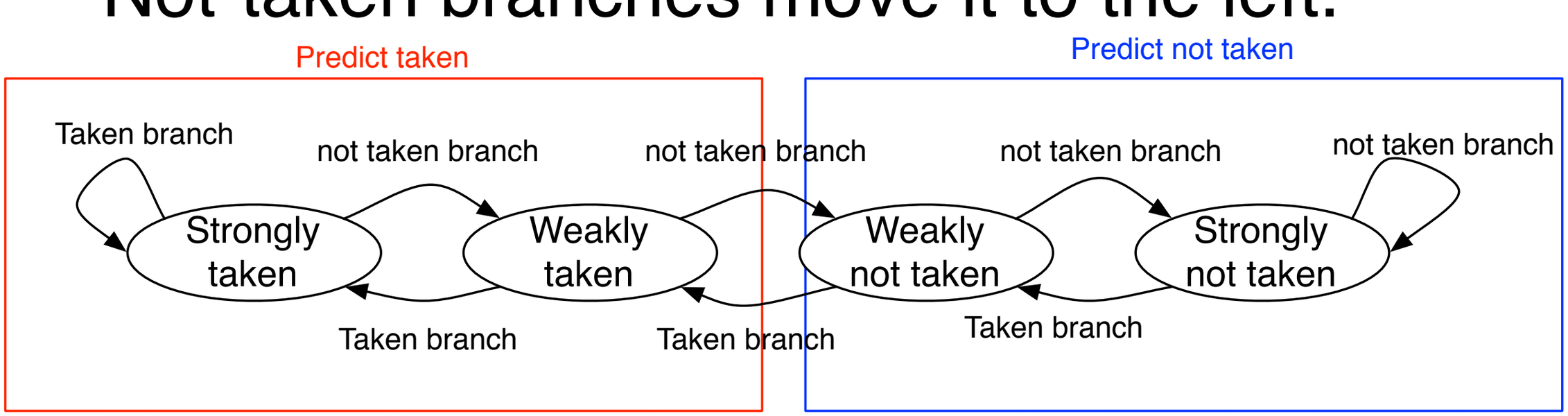
Quiz 6

1. True/False
   a. Squashing instructions and stalling both result in increased CPI.
   b. Squashing can be used to resolve control hazards.
   c. Stalling cannot be used to resolve control hazards.
   d. Static branch predictors use tables of 2-bit counters.
2. Briefly explain why resolving branches in decode is necessary in the MIPS 5-stage pipeline (assuming it does not do branch prediction, and assuming MIPS has one delay slot).
3. Give two examples of why branch behavior is often predictable.
4. If we double the number of pipeline stages in the MIPS 5-stage design by dividing each existing stage in half, what would the new branch delay penalty be (assume branches resolve at the end of decode)?
5. On a scale of 1-10 (1 being completely unfair and 10 being completely fair), how fair was the midterm?

Dynamic prediction 3: A table of counters

- Instead of a single bit, keep two. This gives four possible states

- Taken branches move the state to the right. Not-taken branches move it to the left.



- The predictor waits one prediction before it changes its prediction

Dynamic Prediction 3: A table of counters

```
for(i = 0; i < 10; i++)   {
  for(j = 0; j < 4; j++) {
  }
}
```

- What's the accuracy for the inner loop's branch? (start in weakly taken)

Dynamic Prediction 3: A table of counters

```
for(i = 0; i < 10; i++)   {
  for(j = 0; j < 4; j++) {
  }
}
```

iteration	Actual	state	prediction	new state
1	taken	weakly taken	taken	strongly taken
2	taken	strongly taken	taken	strongly taken
3	taken	strongly taken	taken	strongly taken
4	not taken	strongly taken	taken	weakly taken
1	taken	weakly taken	taken	strongly taken
2	taken	strongly taken	taken	strongly taken
3	taken	strongly taken	taken	strongly taken

- What's the accuracy for the inner loop's branch? (start in weakly taken)

Dynamic Prediction 3: A table of counters

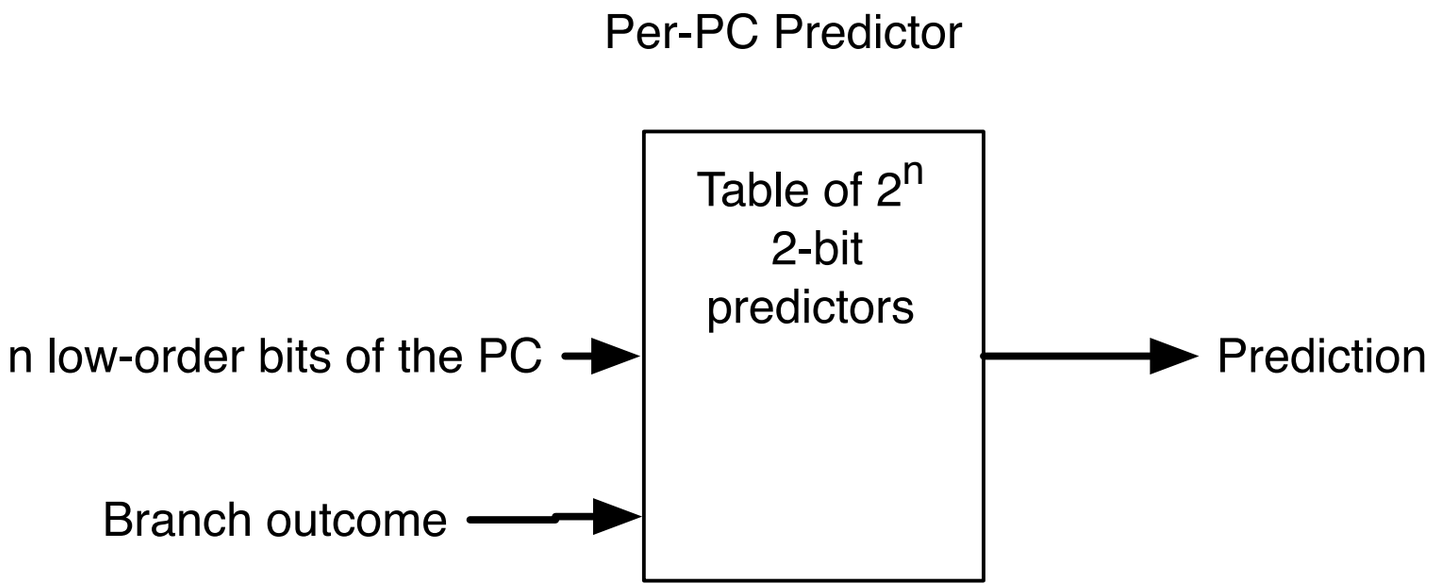
```
for(i = 0; i < 10; i++)   {
  for(j = 0; j < 4; j++) {
  }
}
```

iteration	Actual	state	prediction	new state
1	taken	weakly taken	taken	strongly taken
2	taken	strongly taken	taken	strongly taken
3	taken	strongly taken	taken	strongly taken
4	not taken	strongly taken	taken	weakly taken
1	taken	weakly taken	taken	strongly taken
2	taken	strongly taken	taken	strongly taken
3	taken	strongly taken	taken	strongly taken

- What's the accuracy for the inner loop's branch? (start in weakly taken) 25% or 1 per loop

Two-bit Prediction

- The two bit prediction scheme is used very widely and in many ways.
  - Make a table of 2-bit predictors
  - Devise a way to associate a 2-bit predictor with each dynamic branch
  - Use the 2-bit predictor for each branch to make the prediction.
- In the previous example we associated the predictors with branches using the PC.
  - We'll call this "per-PC" prediction.



Associating 2-bit Predictors with Branches: Using the low-order PC bits

- When is branch behavior predictable?
  - Loops -- for(i = 0; i < 10; i++) {} 9 taken branches, 1 not-taken branch. All 10 are pretty predictable.

    OK -- we miss one per loop
  - Run-time constants
    - Foo(int v,) { for (i = 0; i < 1000; i++) {if (v) {...}}}.
    - The branch is always taken or not taken.

    Good
  - Corollated control
    - a = 10; b = <something usually larger than a >
    - if (a > 10) {}
    - if (b > 10) {}

    Poor -- no help
  - Function calls
    - LibraryFunction() -- Converts to a jr (jump register) instruction, but it's always the same.
    - BaseClass * t; // t is usually a of sub class, SubClass
    - t->SomeVirtualFunction() // will usually call the same function

    Not applicable

Predicting Loop Branches Revisited

```
while (1){
  for(j = 0; j < 3; j++) {
  }
}
```

- What's the pattern we need to identify?

Predicting Loop Branches Revisited

```
while (1){
  for(j = 0; j < 3; j++) {
  }
}
```

iteration	Actual
1	taken
2	taken
3	taken
4	not taken
1	taken
2	taken
3	taken
4	not taken
1	taken
2	taken
3	taken
4	not taken

- What's the pattern we need to identify?

Dynamic prediction 4: Global branch history

- Instead of using the PC to choose the predictor, use a bit vector made up of the previous branch outcomes.

Dynamic prediction 4: Global branch history

- Instead of using the PC to choose the predictor, use a bit vector made up of the previous branch outcomes.

iteration	Actual	Branch history	Steady state prediction
1	taken	11111	
2	taken	11111	
3	taken	11111	
4	not taken	11111	
outer loop branch	taken	11110	taken
1	taken	11101	taken
2	taken	11011	taken
3	taken	10111	taken
4	not taken	, 01111	not taken
outer loop branch	taken	11110	taken
1	taken	11101	taken
2	taken	11011	taken
3	taken	10111	taken
4	not taken	, 01111	not taken
outer loop branch	taken	11110	taken
1	taken	11101	taken
2	taken	11011	taken
3	taken	10111	taken
4	not taken	, 01111	not taken

Dynamic prediction 4: Global branch history

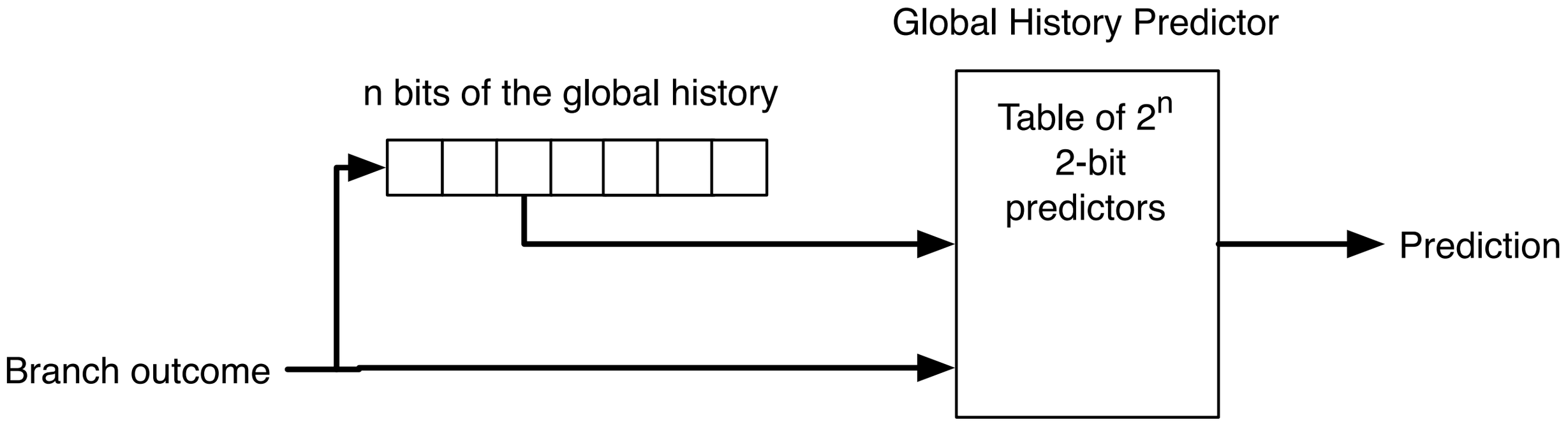
- Instead of using the PC to choose the predictor, use a bit vector made up of the previous branch outcomes.

Nearly perfect

iteration	Actual	Branch history	Steady state prediction
1	taken	11111	
2	taken	11111	
3	taken	11111	
4	not taken	11111	
outer loop branch	taken	11110	taken
1	taken	11101	taken
2	taken	11011	taken
3	taken	10111	taken
4	not taken	, 01111	not taken
outer loop branch	taken	11110	taken
1	taken	11101	taken
2	taken	11011	taken
3	taken	10111	taken
4	not taken	, 01111	not taken
outer loop branch	taken	11110	taken
1	taken	11101	taken
2	taken	11011	taken
3	taken	10111	taken
4	not taken	, 01111	not taken

Dynamic prediction 4: Global branch history

- Instead of using the PC to choose the predictor, use a bit vector made up of the previous branch outcomes.



Dynamic prediction 4: Global branch history

- How long should the history be?

- Imagine N bits of history and a loop that executes K iterations
  - If K <= N, history will do well.
  - If K > N, history will do poorly, since the history register will always be all 1's for the last K-N iterations. We will mis-predict the last branch.

Dynamic prediction 4: Global branch history

- How long should the history be?

  Infinite is a bad choice. We would learn nothing.

- Imagine N bits of history and a loop that executes K iterations
  - If K <= N, history will do well.
  - If K > N, history will do poorly, since the history register will always be all 1's for the last K-N iterations. We will mis-predict the last branch.

Associating Predictors with Branches: Global history

- When is branch behavior predictable?
  - Loops -- for(i = 0; i < 10; i++) {} 9 taken branches, 1 not-taken branch. All 10 are pretty predictable.
  - Run-time constants
    - Foo(int v,) { for (i = 0; i < 1000; i++) {if (v) {...}}}.
    - The branch is always taken or not taken.
  - Corollated control
    - a = 10; b = <something usually larger than a >
    - if (a > 10) {}
    - if (b > 10) {}
  - Function calls
    - LibraryFunction() -- Converts to a jr (jump register) instruction, but it's always the same.
    - BaseClass * t; // t is usually a of sub class, SubClass
    - t->SomeVirtualFunction() // will usually call the same function

Good

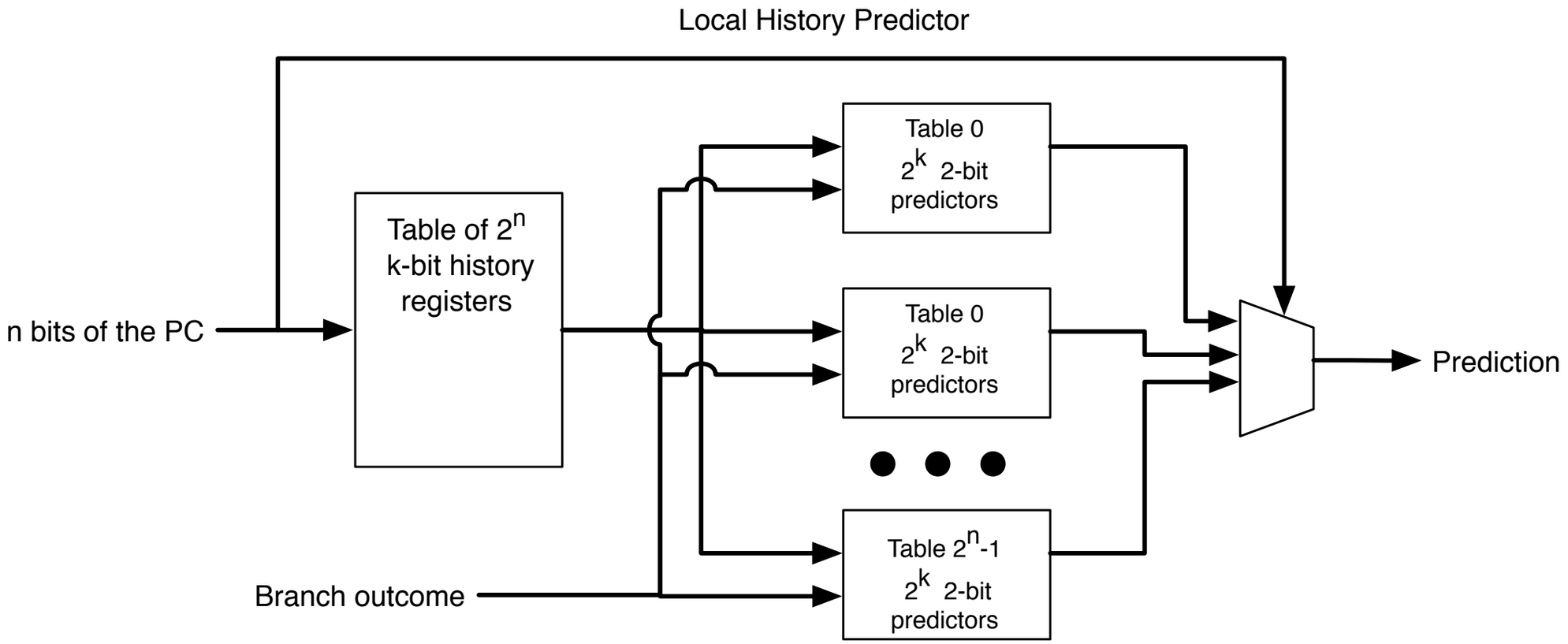
Not so great

Pretty good, as long as the history is not too long

Not applicable

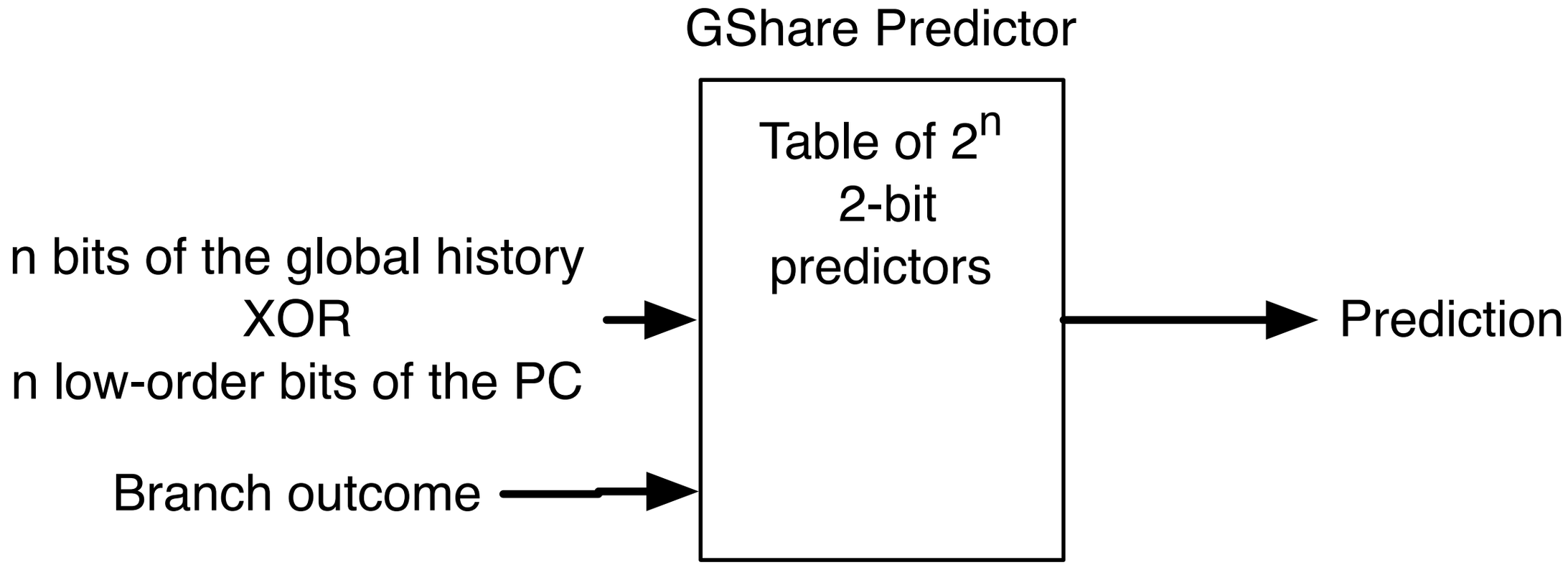
The Local History Predictor

- Use a table of history registers, indexed by the low-order bits of the PC.
- Also use the PC to choose a table, each indexed by the history for that branch.
- For loops this does better than global history.
  - Foo() { for(i = 0; i < 4; i++){ } }.
  - If foo is called from many places, the global history will be polluted, but the local history for the loop's branch will be kept safe.



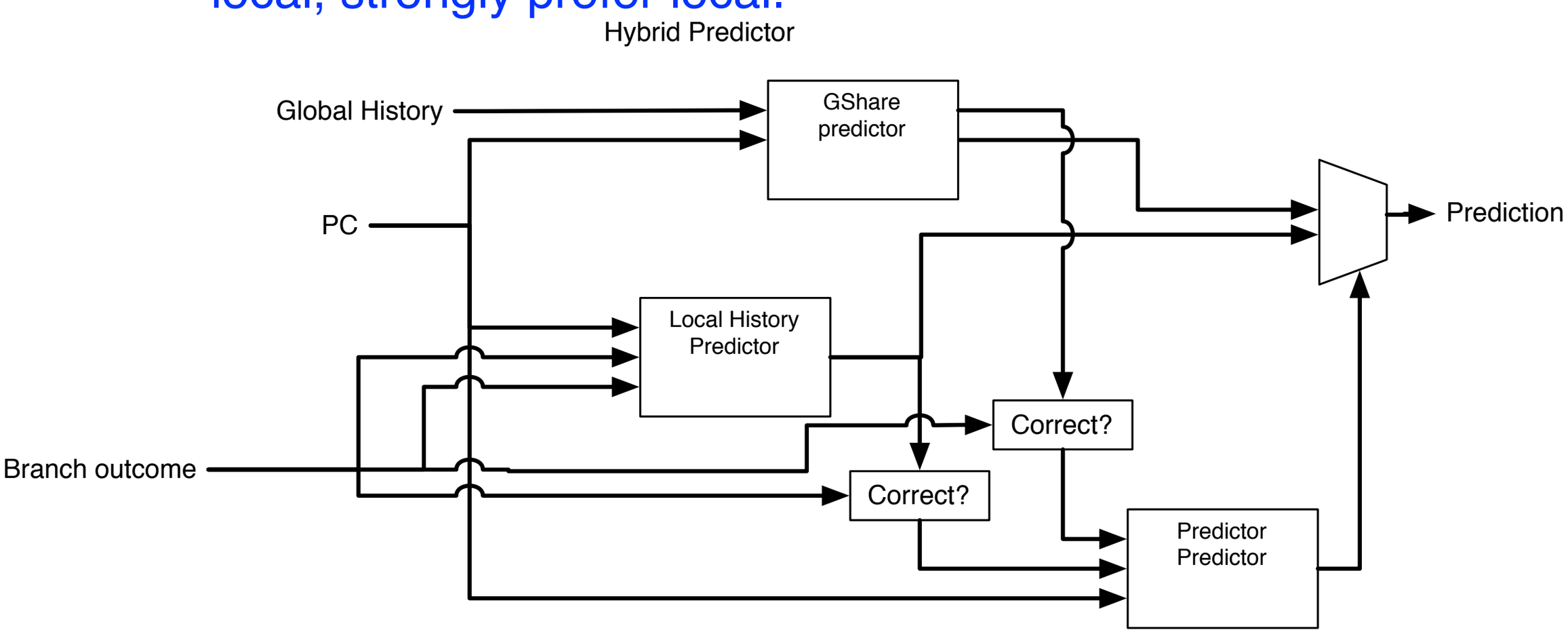
Other Ways of Identifying Branches

- Combine Global History and bits of the PC
  - Gshare predictor
  - Index a of two-bit predictors with the PC XOR Global History.



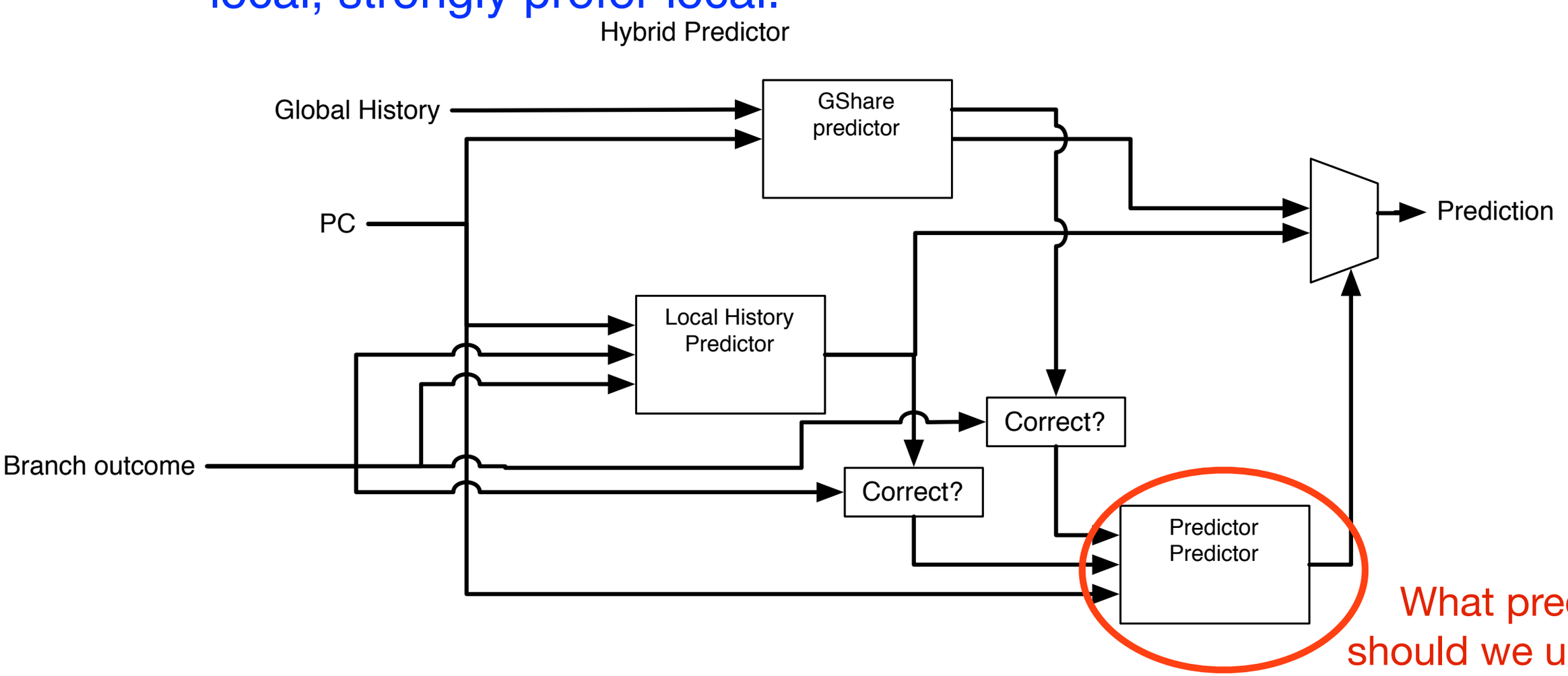
Other Ways of Identifying Branches

- How do we get the best of all possible worlds?
- Build them all, and have a predictor to decide which one to use on a given branch -- The Hybrid (or Tournament) Predictor
  - 2-bit predictor now has different states
  - Strongly prefer GShare, weakly prefer Gshare, weakly prefer local, strongly prefer local.



Other Ways of Identifying Branches

- How do we get the best of all possible worlds?
- Build them all, and have a predictor to decide which one to use on a given branch -- The Hybrid (or Tournament) Predictor
  - 2-bit predictor now has different states
  - Strongly prefer GShare, weakly prefer Gshare, weakly prefer local, strongly prefer local.



The Hybrid Predictor

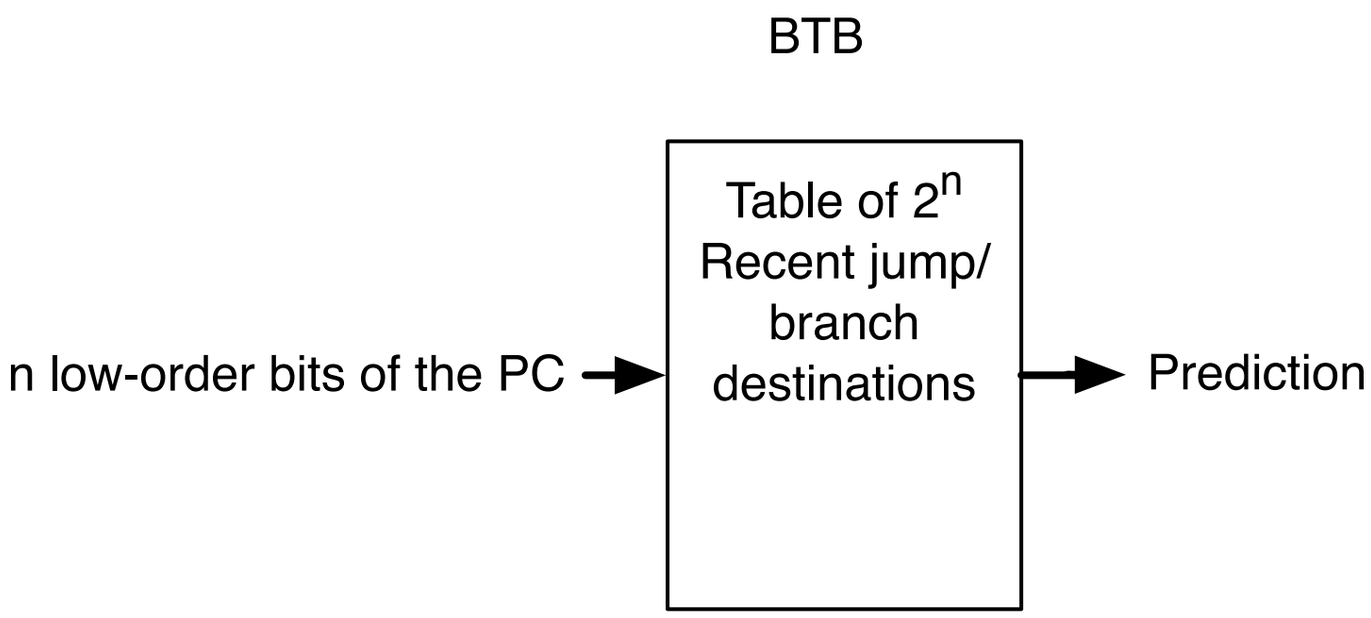
- Loops -- for(i = 0; i < 10; i++) {} 9 taken branches, 1 not-taken branch. All 10 are pretty predictable. — Good
- Run-time constants
  - Foo(int v,) { for (i = 0; i < 1000; i++) {if (v) {...}}}. — Good
  - The branch is always taken or not taken.
- Corollated control
  - a = 10; b = <something usually larger than a > — Good
  - if (a > 10) {}
  - if (b > 10) {}
- Function invocations
  - LibraryFunction() -- Converts to a jr (jump register) instruction, but it's always the same. — Not applicable
  - BaseClass * t; // t is usually a of sub class, SubClass
  - t->SomeVirtualFunction() // will usually call the same function
- Function Returns
  - You have to jump back to where you came from after a function call. — Not applicable

Interference

- Our schemes for associating branches with predictors are imperfect.
- Different branches may map to the same predictor and pollute the predictor.
- This is called “destructive interference”
- Using larger tables will (typically) reduce this effect.

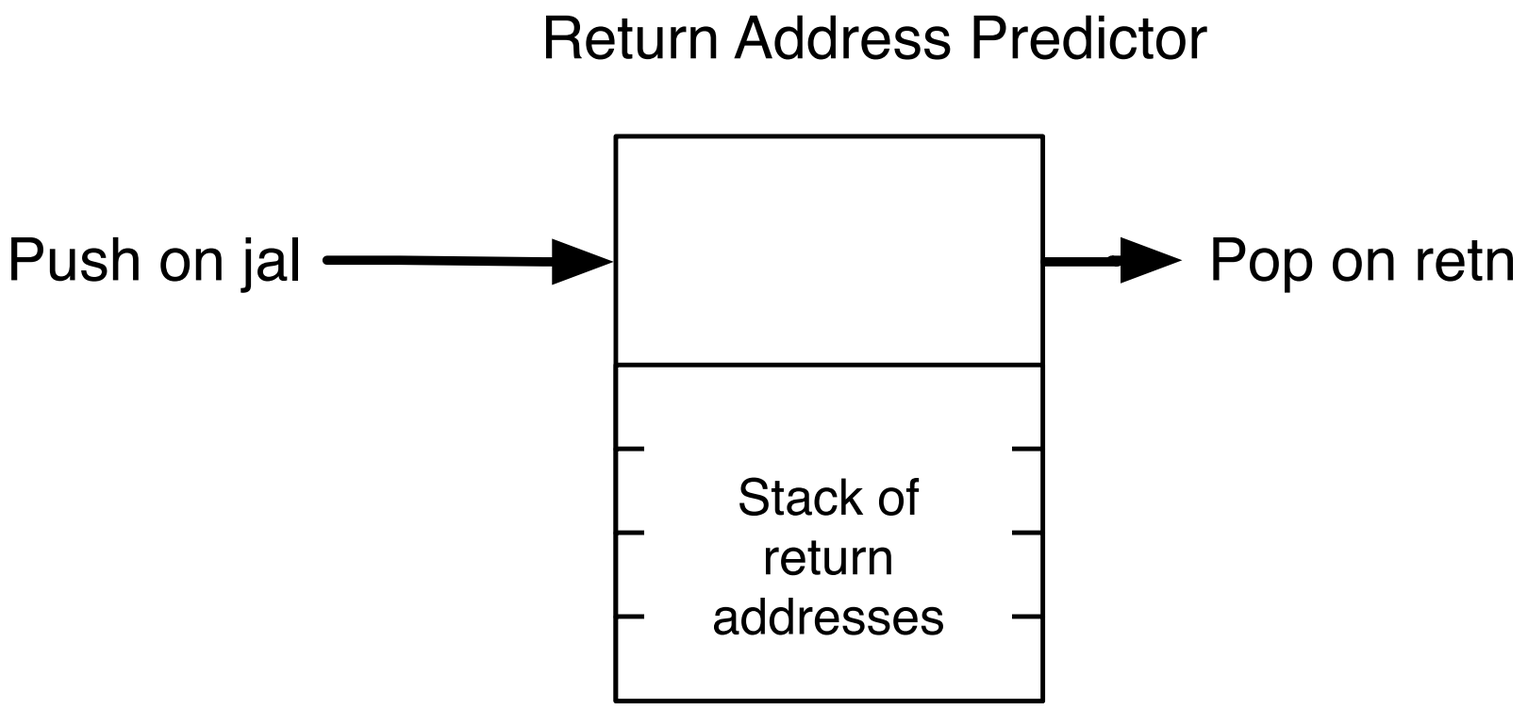
Predicting Function Invocations

- Branch Target Buffers (BTB)
  - Use a table, indexed by PC, that stores the last target of the jump.
  - When you fetch a jump, start executing at the address in the BTB.
  - Update the BTB when you find out the correct destination.
- The BTB is useful for predicting function calls and jump instructions (and some other things, as we will see shortly.)



Predicting Returns

- Function call returns are hard to predict
  - For every call site, the return address is different
  - The BTB will do a poor job, since it's based on PC
- Instead, maintain a "return stack predictor"
  - Keep a stack of return targets
  - `jal` pushes $ra onto the stack
  - Fetch predicts the target for `retn` instruction by popping an address off the stack.
  - Doesn't work in MIPS, because there is no return instruction.



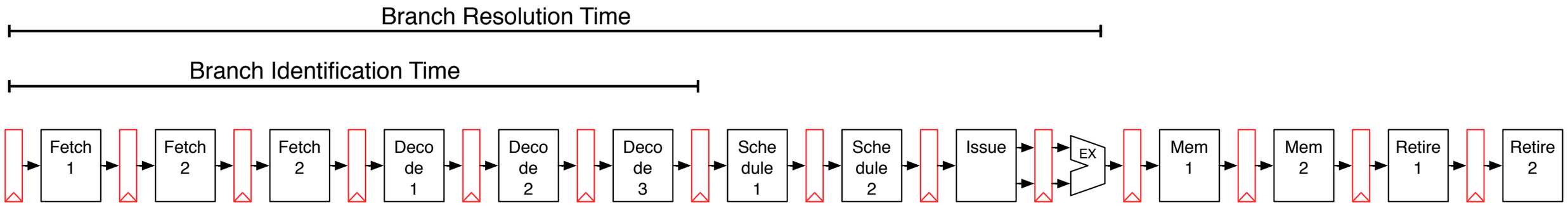
Predicting Returns In MIPS

- The return address predictor doesn't work in MIPS, because there is no return instruction
- How could we fix it?

Predicting Returns In MIPS

- The return address predictor doesn't work in MIPS, because there is no return instruction
- How could we fix it?
  - Add a retn instruction -- it's just jr but with a different opcode so we can tell the difference
  - Build a predictor to choose between the return address predictor and the BTB.

Branch Prediction in Long Pipes



- Modern processor have deep pipelines
- Conditional branch predictors are good, but they can take several cycles to make a prediction
  - What do should we fetch in the meantime?
- Processors will predict each branch multiple times
  - First, use the BTB -- The accuracy may not be great
  - A few cycles later, the conditional branch predictor lets you know if the BTB was probably right or wrong.
  - Several cycles after that, the actual branch direction is known.